Paralelização em GPU de uma Rede Neural de Regressão Geral

Luciano Antonio Siqueira

Trabalho de Conclusão de Curso
de Matemática Aplicada e Computacional
no Instituto de Matemática e Estatística
da Universidade de São Paulo

Supervisor: Prof. Dr. Alfredo Goldman
Co-supervisor: Dr. Marcos Amarís González

São Paulo, Novembro 2018

Paralelização em GPU de uma Rede Neural de Regressão Geral

Banca de TCC:

- Dr. Marcos Amaris González (Co-supervisor - Presidente)
  Universidade de São Paulo (IME - Brasil)
- Prof. Dr. Roberto Hirata
  Universidade de São Paulo (IME - Brasil)
- Prof. Dr. Pedro Peixoto
  Universidade de São Paulo (IME - Brasil)

Resumo

Este trabalho trata da paralelização em GPU do algoritmo **GRNN**, *General Regression Neural Network* ou Rede Neural de Regressão Geral. A partir de um conjunto amostral previamente coletado, o algoritmo constrói uma superfície de regressão multidimensional, utilizada como distribuição de densidade conjunta. Essa distribuição permite calcular o valor esperado de uma variável dependente associado a uma variável independente, ambas possivelmente multidimensionais.

Para avaliar a precisão do método, foram realizados testes comparativos com os mesmos conjuntos de treinamento e teste para a GRNN e para a ferramenta de regressão do TensorFlow. A GRNN se mostrou superior nos cenários analisados, apesar do TensorFlow ser potencialmente superior se for utilizado um maior tempo de treinamento.

A GRNN foi implementada em linguagem C com as extensões NVIDIA CUDA. Foram realizados testes de desempenho em diferentes cenários e em todos eles houve ganho significativo de desempenho, chegando a superar em trezentas vezes a versão sequencial e a obter um ganho dessa mesma ordem em relação à paralelização em CPU.

Os resultados demonstraram que a GRNN é um boa opção onde técnicas de regressão sejam aplicáveis, sobretudo quando o conjunto de dados observados não é estático e não há tempo para executar um método de aprendizado de máquina exige um treinamento mais intenso.

Abstract

The present work deals with the parallelization of the algorithm called *GRNN*: General Regression Neural Network. From a previous collected sample set, the algorithm builds a multidimensional regression surface, applied as a joint density distribution. This distribution allows to compute the expected value of a dependent variable associated with a independent variable, both probable multidimensional.

In order to assess the method accuracy, comparative tests were performed on the same training and test sets using the implemented GRNN algorithm and the TensorFlow regression estimator. The GRNN algorithm performed better in the analysed cases, but TensorFlow can be potentially better if a longer training time is given.

The GRNN algorithm was written using C and the NVIDIA CUDA language extensions. Benchmarks in distinct scenarios shown a relevant gain in performance, up to three hundred times faster than the sequential version, and a same scale improvement over a similar CPU parallelization method.

The achieved results indicate that a parallel GRNN offers a good option when regression techniques are applicable, especially when the observed dataset is not static and there is not much time to run a training-intensive machine learning method.

Conteúdo

Capítulo 1

A Rede Neural de Regressão Geral: GRNN

O algoritmo GRNN (sigla do nome em inglês *General Regression Neural Network*), é capaz de construir uma superfície de regressão que é utilizada para estimar o valor de uma variável dependente a partir de uma variável independente dada. Essa superfície é construída a partir de pares previamente coletadas, que consistem em uma variável independente e uma variável dependente associada.

A rede neural proposta por Specht [1] fornece estimativas de variáveis contínuas e converge para a superfície de regressão, linear ou não-linear. A rede aprende com uma única passagem e possui uma estrutura altamente paralelizável com utilidade em aplicações como aprender a dinâmica de um sistema para sua predição ou controle.

1.1 Valor Esperado da Variável Dependente

A regressão de uma variável dependente, Y, em uma variável independente, X, é o cálculo do valor mais provável de Y para cada valor de X baseando-se num número finito de medições prévias (possivelmente imprecisas) de X e seus valores Y correspondentes. As variáveis X e Y podem ser vetoriais, particularmente $\mathbf{x} \in \mathbb{R}^p$.

Como demonstrado no Apêndice A, a estimativa para Y é obtida calculando a esperança da variável aleatória Y dada $X = x$ usando uma função $f(\mathbf{x}, y)$ contínua que representa a densidade da probabilidade conjunta de $\mathbf{X}$ e Y. O vetor $\mathbf{x}$ é um vetor específico da variável aleatória $\mathbf{X}$. O valor esperado para Y dada $\mathbf{X} = \mathbf{x}$ (regressão de Y em $\mathbf{X} = \mathbf{x}$) é dado por

$$\mathbb{E}[Y|\mathbf{X} = \mathbf{x}] = \frac{\int_{-\infty}^{\infty} y f(x, y)\, dy}{\int_{-\infty}^{\infty} f(x, y)\, dy}. \tag{1.1}$$

Quando desconhecida, a função de densidade pode ser substituída por uma função

$\hat{f}(\mathbf{x}, y)$ que estima a probabilidade baseando-se em amostras obtidas de $\mathbf{X}$ e Y. Substituindo na equação da esperança (1.1), se obtém a função $\hat{\mathbb{E}}[Y|\mathbf{X} = \mathbf{x}]$ que estima o valor esperado:

$$\hat{\mathbb{E}}[Y|\mathbf{X} = \mathbf{x}] = \frac{\sum_{i=1}^{n} y^{(i)} \exp\left(-\frac{d^{(i)}}{2\sigma^2}\right)}{\sum_{i=1}^{n} \exp\left(-\frac{d^{(i)}}{2\sigma^2}\right)} \tag{1.2}$$

onde n é o número de amostras e $d^{(i)} = d^{(i)}(\mathbf{x}) = (\mathbf{x} - \mathbf{x}^{(i)})^T(\mathbf{x} - \mathbf{x}^{(i)})$ é uma função escalar de distância entre o vetor de entrada $\mathbf{x}$ e o vetor $\mathbf{x}^{(i)}$ do par amostral $(\mathbf{x}^{(i)}, y^{(i)})$.

Parzen [2] e Cacoullos [3] demonstraram que a forma (1.2) converge assintoticamente para (1.1) em todos os pontos $(\mathbf{x}, y)$ onde a função de densidade é contínua, desde que $\sigma = \sigma(n)$ seja definido como uma função decrescente de n tal que

$$\lim_{n\to\infty} \sigma(n) = 0 \quad \text{e} \quad \lim_{n\to\infty} n\sigma(n)^p = \infty \tag{1.3}$$

onde p é o número de dimensões da variável aleatória $\mathbf{X}$.

Existem diferentes funções $\sigma(n)$ que respeitam os critérios (1.3). Uma escolha natural é a recíproca da função logarítmica:

$$\sigma(n) = \frac{1}{\log(n)}. \tag{1.4}$$

1.2 Parâmetro da Regressão

A função (1.4) respeita os critérios (1.3) e oferece boas aproximações para diferentes conjuntos de dados. Contudo, pode ser conveniente utilizar um parâmetro escalar para conseguir melhores aproximações em um conjunto específico de dados. Incluindo um segundo parâmetro para esse escalar, a função σ fica dada por

$$\sigma(n, s) = \frac{s}{\log(n)} \tag{1.5}$$

onde o parâmetro s é um número real positivo "próximo" de 1. Um parâmetro s menor que 1 reduz a influência local de cada amostra em particular, aumentando a definição da superfície de regressão. Já um parâmetro s maior que 1 suaviza a superfície de regressão, aumentando a abrangência de cada amostra. Uma superfície de regressão mais ou menos definida depende muito da quantidade e disposição das amostras no conjunto de treinamento. Em geral, testes demonstraram que um bom ponto de partida são valores de s entre $\frac{1}{5}$ e $\frac{1}{2}$.

Essa simplicidade na parametrização do modelo, que exige o parâmetro único σ, é uma vantagem quando se quer verificar rapidamente a viabilidade de um método para aprendizado, predição e controle (exemplos em [4], [5] e [6]). Sendo capaz de

fornecer resultados rapidamente, a GRNN pode ser analisada, adotada ou descartada sem dificuldade ou prejuízo significativo de tempo.

1.3 Paralelização da GRNN

Como observado por Specht [1], o grande número de operações é o maior obstáculo prático para a implementação eficiente da GRNN. Contudo, o próprio autor destaca que o algoritmo possui uma estrutura paralela inerente e sua ineficiência seria mitigada se utilizado hardware especializado para processamento paralelo.

A versatilidade com que a GRNN pode ser particionada permite sua implementação utilizando programação paralela de uso geral. Das muitas tecnologias de computação paralela disponíveis, a escolha foi a implementação usando as extensões CUDA em placas de GPU NVIDIA.

Como ilustrado por Owens et al. [7], o tipo de paralelismo oferecido pela GPU é apropriado quando se quer executar uma mesma operação em elementos distintos na memória – *SIMD*, *Single Instruction Multiple Data* – que é exatamente o caso da GRNN (ilustrada na Figura 1.1).

Por si só, contudo, a paralelização em GPU pode não trazer vantagens significativas se implementada ingenuamente. Como observado por Ryoo et al. [8], a programação eficiente em CUDA deve seguir alguns princípios básicos:

1. Construir *threads* simples para evitar a latência na alocação de novos threads.
2. Otimizar o uso da memória no Chip (*shared memory*) para reduzir o acesso redundante à memória geral da GPU.
3. Garantir que as leituras à memória geral a partir dos threads seja feita de modo agrupado para reduzir o número de instruções de leituras.
4. Quando necessário, fazer uso da sincronização para trocar informações entre threads de um mesmo bloco.

Esses princípios são observados nas técnicas demonstradas por Harris em [9], [10], [11] e [12], que foram tomadas como base para implementação. No caso da GRNN, o primeiro e o terceiro princípio são os essenciais: cada estimativa para a variável dependente associada a uma variável independente informada requer operações realizadas em cada uma das amostras no conjunto de treinamento. Tendo em vista que as operações envolvendo cada amostra são independentes umas das outras, foram escolhidas essas operações como os elementos que serão executados em paralelo.

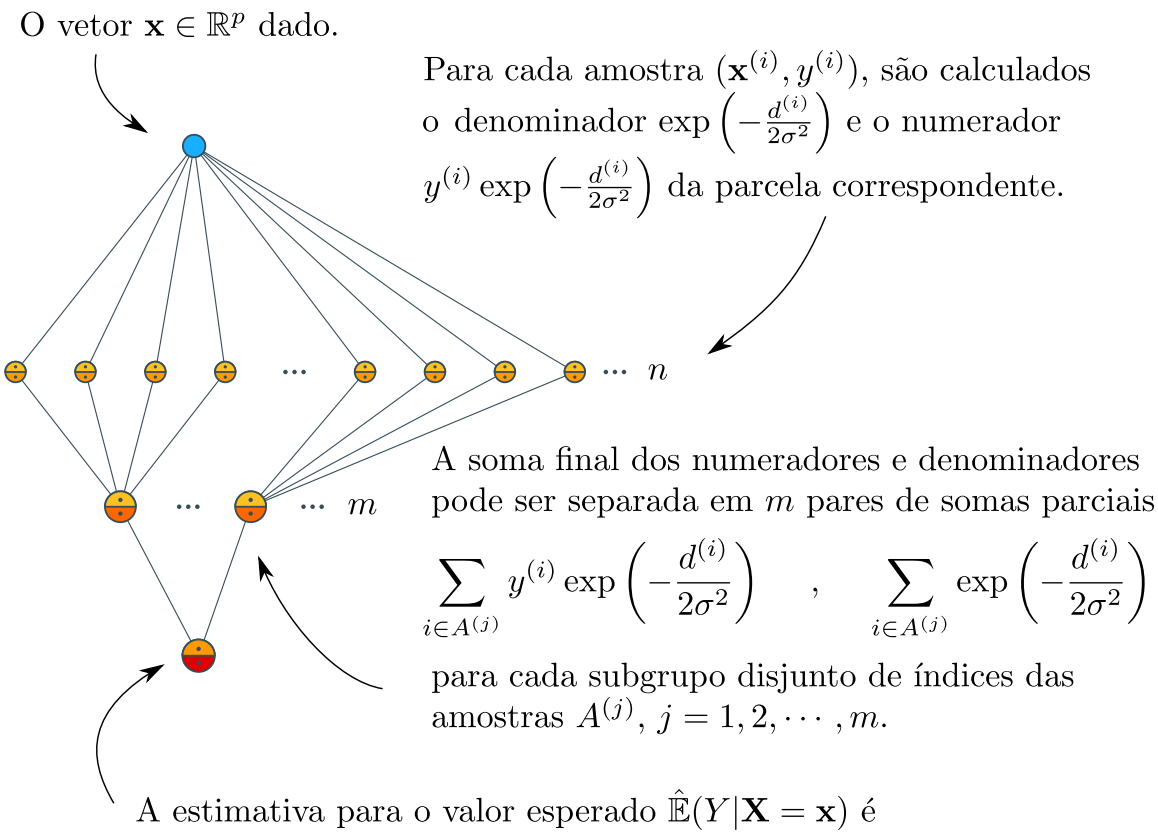

Figura 1.1: Estrutura de uma Rede Neural de Regressão Geral. O valor $d^{(i)} = (\mathbf{x} - \mathbf{x}^{(i)})^T(\mathbf{x} - \mathbf{x}^{(i)})$ é uma medida de distância entre $\mathbf{x}$ e a i-ésima amostra $\mathbf{x}^{(i)}$. O valor σ está associado à variância da distribuição gaussiana subjacente. Se Y for uma variável aleatória dependente multivariada, então o mesmo procedimento é realizado para cada componente $\mathbf{y}_k^{(i)}$.

1.4 Unidades de Processamento Gráfico (GPUs)

As unidades de processamento gráfico inicialmente possuíam apenas função gráfica, mas rapidamente seu potencial foi percebido e novas arquiteturas evoluíram dando suporte ao processamento de propósito geral *(GPGPU)*. O paralelismo explorado pelas GPUs é do tipo *Single Instruction Multiple Data* (SIMD), mas comumente tem sido utilizado uma denominação derivada, nomeada *Single Instruction Multiple Threads* (SIMT), onde milhões de *threads* são lançados nos núcleos presentes em uma GPU.

No tipo de paralelismo usado pelo SIMT, a mesma instrução é executada simultaneamente por diferentes linhas de execução. Assim, no modelo de programação SIMT uma aplicação lança um conjunto de linhas de execução as quais executarão todas a mesma instrução, e essas linhas de execução são programadas de forma dinâmica em um paralelismo do tipo SIMD.

As GPUs são compostas de milhares de núcleos *(cores)* simples que executam o mesmo código através de milhões de *threads* concorrentes. O que leva a uma abordagem que difere do modelo tradicional de processadores *multicore*. Figura 1.2 mostra um esquema genérico de um sistema heterogêneo composto de uma CPU e de uma GPU.

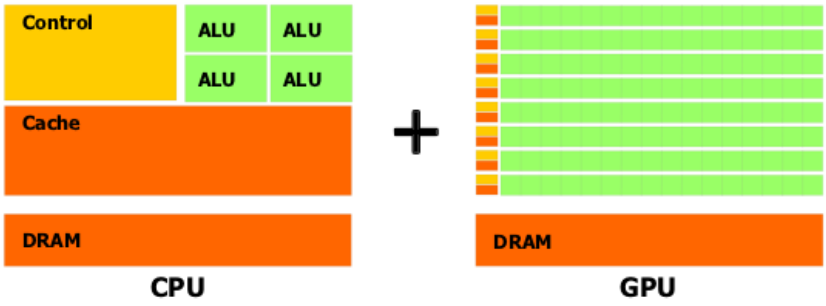

Figura 1.2: Sistema Heterogêneo CPU-GPU.

Os núcleos que compõem um sistema *multicore* são CPUs completas que possuem poderosas unidades de controle e de execução, que possibilitam a execução de instruções em paralelo e fora de ordem, além de contarem com uma especializada hierarquia de cache. Já as GPUs contam com unidades de controle e de execução mais simples, onde uma unidade de despacho envia apenas uma instrução para um conjunto de núcleos que a executarão em ordem.

1.4.1 *Compute Unified Device Architecture* (CUDA)

A plataforma de computação paralela CUDA *(Compute Unified Device Architecture)* foi desenvolvida pela NVIDIA em 2006. Desde o seu surgimento, diversos trabalhos têm se dedicado à exploração dessa plataforma na execução de aplicações paralelas.

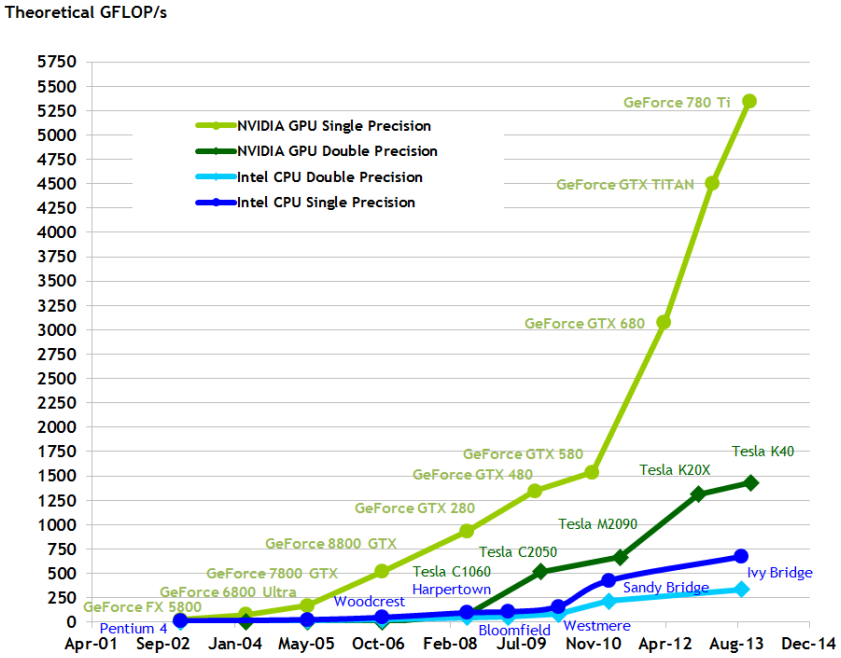

Figura 1.3: Evolução de desempenho das GPUs. Fonte: [13]

O conceito fundamental da execução paralela em dispositivos CUDA está associado ao uso de *threads* que são executadas em um dispositivo separado, a GPU, que trabalha como um acelerador ou coprocessador do computador anfitrião. Desta maneira, o código principal é executado na CPU, a qual faz chamadas a funções denominadas *kernels* que são executadas pela GPU, conforme apresentado na Figura 1.4.

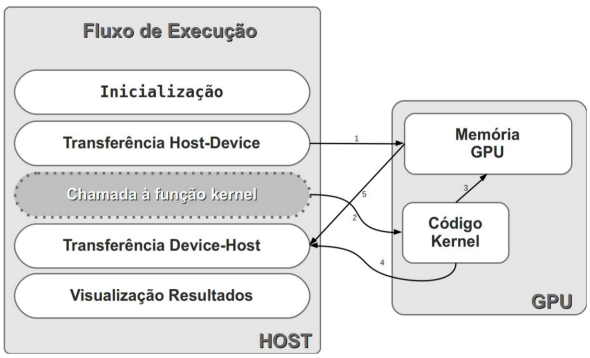

Figura 1.4: Modelo Clássico de Fluxo de Execução centrado na CPU

Um *kernel* é um conceito estendido de uma função C. Porém, ao contrário desta, quando uma chamada a um *kernel* é feita, são executadas t instâncias paralelas por t *threads* CUDA. Todas as *threads* executam uma cópia do mesmo código definido na declaração do corpo da função *kernel*.

Uma função é definida como *kernel* usando-se em sua declaração o modificador `__global__`. Uma chamada a um *kernel* distingue-se de uma chamada a uma função comum pela especificação do número de *threads* que a executarão e como será a or-

ganização do arranjo formado por essas *threads*. A especificação desse arranjo é feita na chamada usando a notação `<<<N, M>>>` entre o nome do *kernel* e da sua lista de parâmetros. Os valores aqui representados por N e M são parâmetros para o *Runtime* da plataforma CUDA. Eles descrevem como deve ser a organização das *threads* criadas pelo lançamento da execução do *kernel*. O N indica o número de blocos paralelos pertencentes ao *grid* e o M o número de *threads* em cada bloco. O que resulta em uma matriz ou até mesmo um cubo de *threads* dependendo das dimensões permitidas pela GPU utilizada.

Os blocos de *threads* são distribuídos nos multiprocessadores da GPU, onde são divididos em *warps* e enviados aos processadores por meio do agendador de *warps*. Linhas de execução de um mesmo *warp* rodam concorrentemente as mesmas instruções. Quando o código tem instruções de condicional, as instruções executadas podem tomar diferentes caminhos. Conforme o resultado da condicional, cada *branch* (ramificação) do *warp* é executada uma de cada vez, mantendo um grupo de processadores sem uso. Esse fenômeno, conhecido como divergência de código.

O modelo de programação CUDA utiliza uma estrutura, ou forma de organizar o contexto de execução, em: *grids*, blocos e *threads*. Neste modelo, um arranjo hierárquico de *threads* é formado com as *threads* que são agrupadas em blocos e estes agrupados em um *grid*. O resultado desse arranjo é um *kernel* sendo executado como um *grid* de blocos de *threads*, cada *thread* executando uma cópia do mesmo código.

As dimensões para um *grid* são limitadas e variam conforme o dispositivo, bem como o número de *threads* em um bloco não pode ser excedido. A informação do número máximo de *threads* suportado está disponível no campo `maxThreadsPerBlock` do registro de informações do dispositivo, que pode ser recuperado chamando a função `cudaGetDeviceProperties(parâmetros)`.

Com um arranjo estrutural de computação com múltiplos blocos e múltiplas *threads*, a indexação será análoga ao método para conversões entre espaço bidimensional e o espaço linear, conforme apresentado na Figura 1.5.

1.5 Paralelização da GRNN em CUDA

O Grid de execução dos blocos de threads foi montado de modo que cada thread corresponda a uma amostra do conjunto de treinamento. Se assume que o conjunto de treinamento pode ser tão grande quanto a capacidade total de memória global disponível no dispositivo, portanto é muito importante que os acessos à memória global sejam agrupados para reduzir o número de instruções de leitura.

Além da leitura agrupada dos dados na memória global, o uso da memória compartilhada no bloco pode melhorar o desempenho. No caso da abordagem adotada, os dados

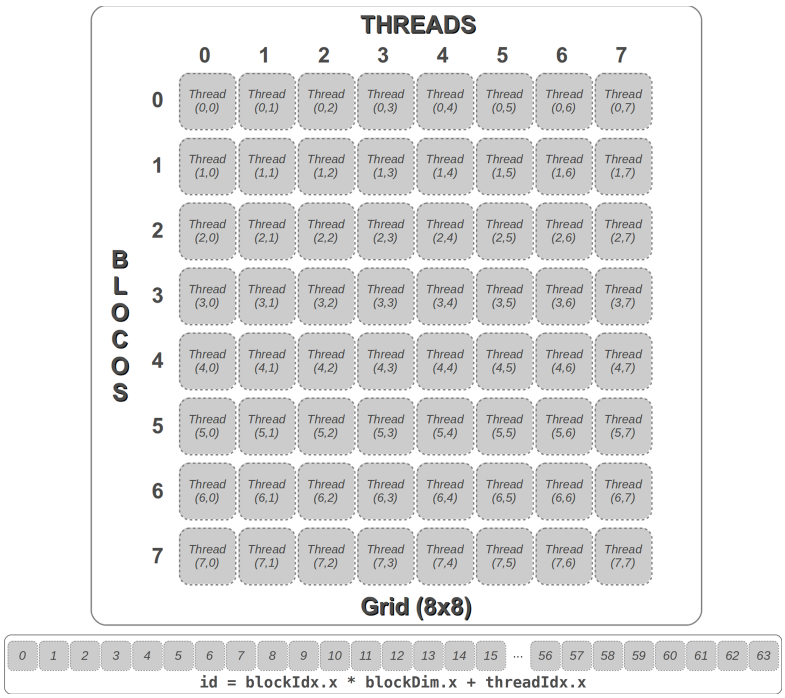

Figura 1.5: Tradução de um *grid* para endereçamento linear

utilizados por cada thread não são compartilhados entre si. Contudo, cada componente da variável dependente é tratado como uma variável dependente unidimensional e requer um mesmo fator utilizado pelas outras componentes. Especulou-se que uma pequena melhoria poderia ser obtida ao armazenar esse fator na memória compartilhada, quando a variável independente possui mais de uma dimensão. Essa melhoria não foi identificada nos testes realizados, provavelmente em função do fator comum ser exclusivo a cada thread e não precisar ser compartilhado com outros threads. Por isso, todos os dados manipulados por cada thread estão nos registradores do multiprocessador ou diretamente na memória global, cujas leituras são armazenados no cache L1[1] em segmentos de 128 bytes e no cache L2 em segmentos de 32 bytes.

Foi definido que a quantidade de threads alocados por bloco deve ser sempre a maior permitida pelo dispositivo, por padrão 1024 threads por bloco. Esse princípio favorece a leitura simultânea dos dados agrupados e permite o cálculo mais eficiente de cada parcela numerador/denominador associada a um bloco no cálculo do quociente final.

Usando como exemplo uma amostra com variável independente de dimensão três e uma variável dependente de dimensão dois (Figura 1.6), um conjunto de treinamento com oito amostras seria agrupado na memória como exemplificado na Figura 1.7.

Se já estiverem dispostos dessa forma na memória do computador anfitrião, um ganho significativo de tempo será obtido utilizando o recurso de memória mapeada do CUDA.

[1]Por padrão, apenas o cache L2 é utilizado para armazenar os acessos à memória global nas gerações de GPU Kepler e Maxwell. Para estas, o cache na memória L1 é ativado com as opções `-Xptxas -dlcm=ca` passadas ao compilador `nvcc`.

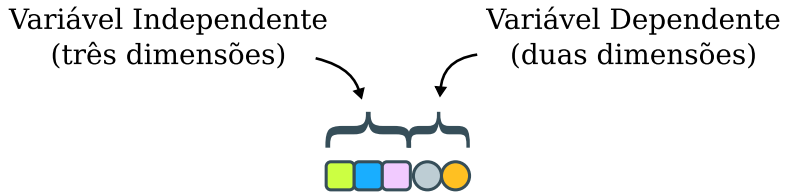

Figura 1.6: Uma estrutura básica para a amostra de treinamento, com três dimensões para a variável independente e duas dimensões para a variável dependente.

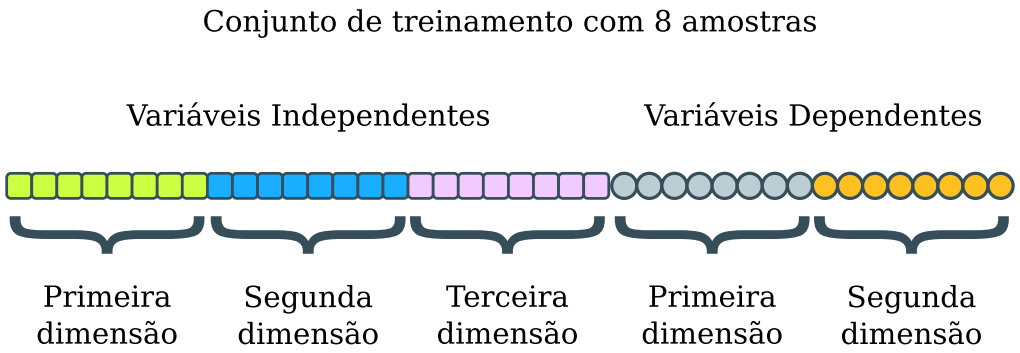

Figura 1.7: O conjunto de treinamento é arranjado na memória em grupos por dimensão. A quantidade de valores em cada grupo corresponde ao total de amostras no conjunto de treinamento

Com ele, a transferência de grandes porções contínuas de memória serão feitas sem que haja paginação, agilizando a cópia do conjunto de treinamento para a memória global do dispositivo.

Cada thread faz a leitura de sua amostra correspondente, um componente por vez, usufruindo das leituras agrupadas proporcionada pelos dados dispostos contiguamente.

É possível observar que os threads sempre requerem um valor na mesma dimensão que os demais threads simultâneos no bloco, porém cada um em sua respectiva amostra no conjunto de treinamento. A Figura 1.8 ilustra a sequência de leituras a cada componente do par amostral na memória global feitas pelos threads em um mesmo bloco.

Depois de efetuar a operação na amostra, um thread produz dois valores que serão somados ao numerador e ao denominador do quociente final da estimativa.

Supondo um número máximo de threads por bloco de apenas quatro threads, o processo de calcular as parciais (numerador e denominador) para o bloco é ilustrado na Figura 1.9.

Todos os threads num mesmo bloco contribuem para uma mesma parcelas do numerador e uma mesma parcela do denominador associadas ao bloco, o que pode gerar conflitos de leitura/escrita em casos de acesso simultâneo. Para contornar esse problema, são utilizadas as funções atômicas do CUDA para efetuar as somas correspondentes a cada bloco. Essas funções garantem que cada thread fará seu acréscimo na parcial do bloco sem conflitar com outro thread simultâneo e é uma estratégia que oferece melhor

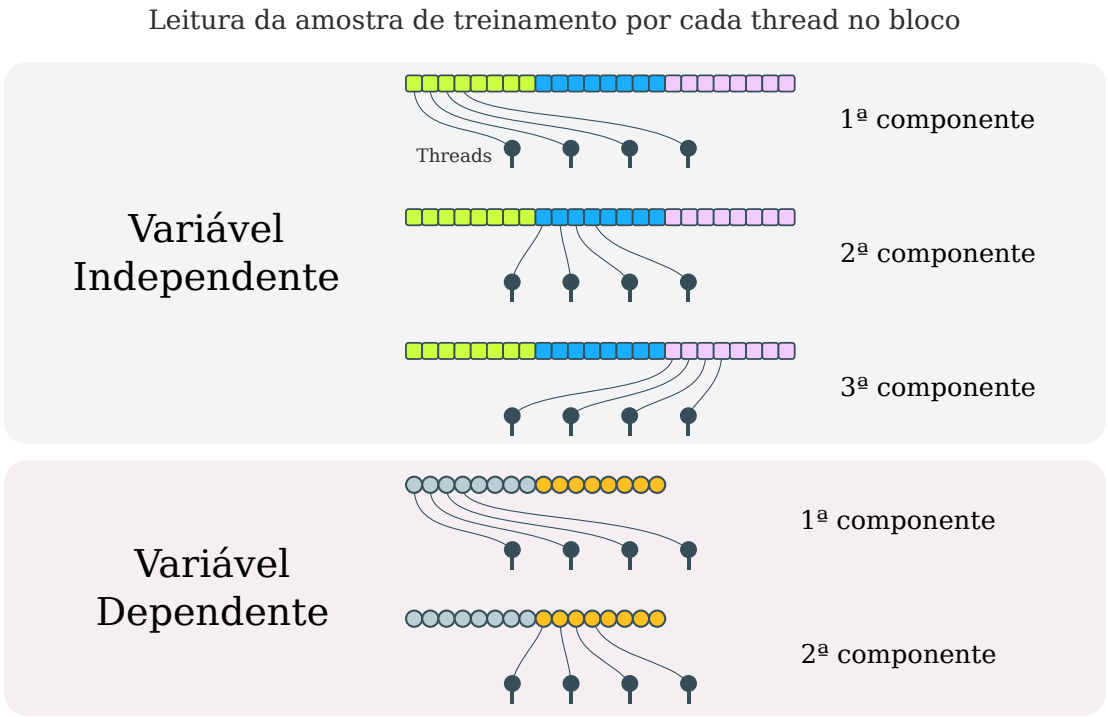

Figura 1.8: Diagrama para um bloco com quatro threads. Cada thread no bloco está associado a um par amostral no conjunto de treinamento. A leitura simultânea é feita na primeira dimensão, depois na segunda dimensão e assim sucessivamente durante a execução do thread.

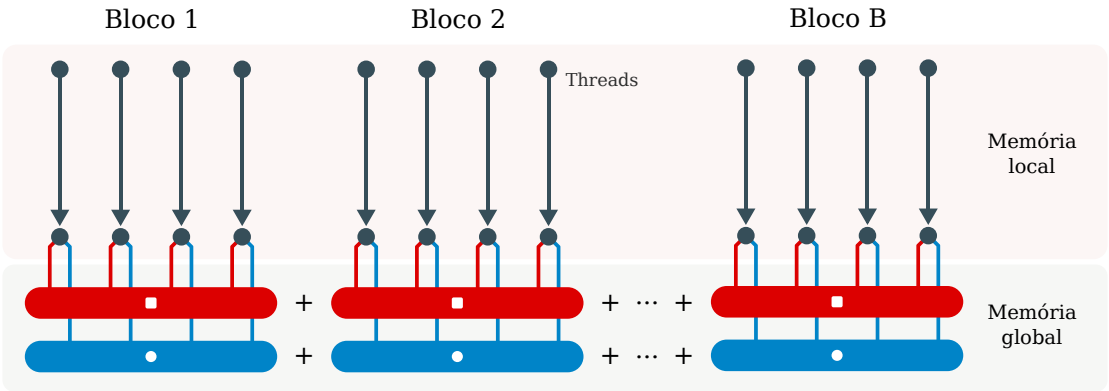

Figura 1.9: O par de resultados parciais (numerador e denominador) gerado por cada bloco é armazenado na memória global e será consolidado após o término da execução do grid.

desempenho frente a uma soma sequencial posterior de cada parcela dos threads.

A estimativa final é calculada após a execução do grid. As parcelas associadas a cada bloco são somadas no numerador e denominador final e a estimativa é obtida com o quociente (1.6):

$$\hat{\mathbb{E}}[Y|\mathbf{X}=\mathbf{x}] = \frac{\sum_{i=1}^{B} b_n^{(i)}}{\sum_{i=1}^{B} b_d^{(i)}} \tag{1.6}$$

A quantidade de parcelas B das somatórias no quociente (1.6) corresponde ao número de blocos. O valor $b_n^{(i)}$ corresponde à soma parcial do numerador computada pelo i-ésimo bloco e o valor $b_d^{(i)}$ corresponde à soma parcial do denominador computada pelo i-ésimo bloco.

Capítulo 2

Análise Comparada de Precisão

A premissa para a construção da rede neural aqui tratada é a existência de um conjunto de resultados – o conjunto de treinamento – já observados, a partir dos quais a rede será capaz de estimar resultados ainda não observados.

Para avaliar o erro médio da estimativa, um segundo conjunto de soluções conhecidas – o conjunto de teste – foi gerado. Após a rede produzir uma estimativa da variável dependente para cada variável independente deste conjunto, é verificada a sua diferença em relação à solução conhecida correspondente. A medida de erro escolhida para todos os erros foi a *RMSE*, ou seja, a raiz-quadrada da média dos quadrados dos erros.

O tamanho do erro é diretamente afetado pelo tipo do problema, quantidade e qualidade do conjunto de treinamento, além de ser de interpretação subjetiva e que depende do propósito da aplicação. Uma maneira de interpretar e analisar qualitativamente o erro é compara-lo com o erro produzido por outro método semelhante, utilizando os mesmos conjuntos de dados. O *TensorFlow* é uma ferramenta para deep learning bastante consolidada e por isso foi escolhida para a realização dos testes comparativos. O TensorFlow oferece um estimador específico para problemas de regressão, chamado *DNNRegressor*, que foi utilizado para a comparação.

Dois testes foram elaborados. O primeiro envolve a construção do fractal Mandelbrot a partir de um conjunto finito de pontos do fractal. A escolha por esse problema se deve a dificuldade em produzir uma estimativa precisa com uma superfície de regressão bastante irregular. O segundo teste estima as solução para problemas de valor inicial em envolvendo a equação diferencial parabólica (equação do calor). Por ser uma transformação linear (vide Apêndice B), é esperado que a superfície de regressão para esse problema seja mais suave e as estimativas mais precisas.

2.1 Rede Neural no TensorFlow

Apesar de terem a mesma finalidade, a maneira como a GRNN e o estimador DNNRegressor do TensorFlow trabalham é bastante diferente. Enquanto que a GRNN percorre todo o conjunto de treinamento sempre que produz uma estimativa, o TensorFlow usa o conjunto de treinamento para atualizar os parâmetros em operadores que mais tarde serão utilizados para produzir a estimativa.

Essa atualização de parâmetros do TensorFlow deve ser feita repetidamente para se obter uma rede neural que produza estimativas satisfatórias. Ao conjunto dessas repetições dá-se o nome *treinamento*. Além disso, a configuração das camadas internas da rede neural interfere no tempo de treinamento e na produção da estimativa.

O treinamento da rede neural consiste em realizar vários passos, ou *steps*, em lotes, ou *batches*, que são um subconjunto do conjunto de treinamento. À repetição desse processo se dá o nome epoca, ou *epoch*. Essa abordagem busca não ajustar perfeitamente a rede neural ao conjunto de treinamento (*overfitting*), de modo que produza boas estimativas quando analisando uma variável não observada. As melhores quantidades de passos, lotes e épocas estão diretamente associadas aos dados envolvidos e podem variar bastante de um problema para outro.

Diferentes abordagens foram testadas e os resultados aqui apresentados correspondem aos melhores que foram obtidos. Contudo, é importante lembrar que outras configurações da estrutura da rede neural e do método de treinamento podem chegar a resultados melhores (ou piores). O modelo de implementação da rede neural no TensorFlow é bastante distinta da GRNN, por isso foi formulada uma configuração que respeitasse as características desse modelo e que fosse simples o suficiente para permitir uma comparação razoável com a GRNN.

2.2 Conjunto Mandelbrot

O conjunto de Mandelbrot é um fractal definido como o conjunto de pontos c no plano complexo para o qual a sequência definida recursivamente

$$z_0 = 0$$
$$z_{n+1} = z_n^2 + c$$

não tende ao infinito. Existem muitos métodos para identificar os pontos do conjunto Mandelbrot, mas todos passam por iterar os pontos candidatos.

O TensorFlow e a GRNN foram utilizados para identificar se um ponto no plano complexo presente no conjunto de teste pertence ao conjunto Mandelbrot. Esse problema foi escolhido exatamente pela dificuldade em representar o fractal numa superfície de regressão.

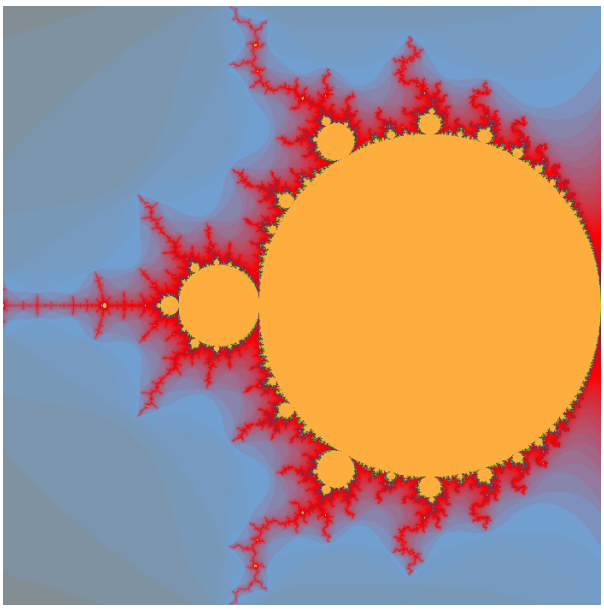

Figura 2.1: Os pontos de treinamento para o conjunto Mandelbrot correspondem a um grid discreto compreendendo o semiplano $Re \in [-1,625, -0,75]$, $Im \in [-0,4375, 0,4375]$. As cores estão associadas ao número de iterações que o algoritmo usou para identificar se o ponto pertence ao conjunto. Esse número também foi utilizado como a variável dependente na regressão.

Para o conjunto de treinamento, foi gerado um semiplano discretizado pelo algoritmo recursivo e a cada ponto nesse semiplano está associado o número de iterações. Geralmente este número está associado a uma cor, daí surgem as conhecidas formas fractais associadas ao Mandelbrot (Figura 2.1). Para gerar o conjunto de treinamento, o maior número de iterações foi fixado em 1000. Além disso, a variável dependente (o número de iterações) foi dividida por 1000, normalizando-a entre 0 e 1.

O conjunto de teste foi um conjunto de 8192 pontos aleatórios contido no mesmo semiplano do conjunto de treinamento. Devido à discretização, os pontos no conjunto de teste não se repetem no conjunto de treinamento.

2.2.1 Erro Comparado

O erro apresentado pela GRNN foi utilizado como denominador em uma quociente cujo numerador é o erro apresentado pelo estimador do TensorFlow. Dessa forma, o valor do quociente será superior a 1 se o erro no TensorFlow for maior que o erro da GRNN e inferior a 1 se o erro no TensorFlow for menor que o erro da GRNN.

Três conjuntos de treinamento de diferentes tamanhos foram produzidos para o teste. Todos os conjuntos correspondem a mesma área exibida na Figura 2.1, mas com uma densidade de 300x300 (90000 pontos), 700x700 (490000 pontos) e 1000x1000 (1000000 pontos). Os melhores resultados com o TensorFlow foram obtidos após o treinamento em lotes de 500 amostras com 150 passos, repetidos em 20 épocas.

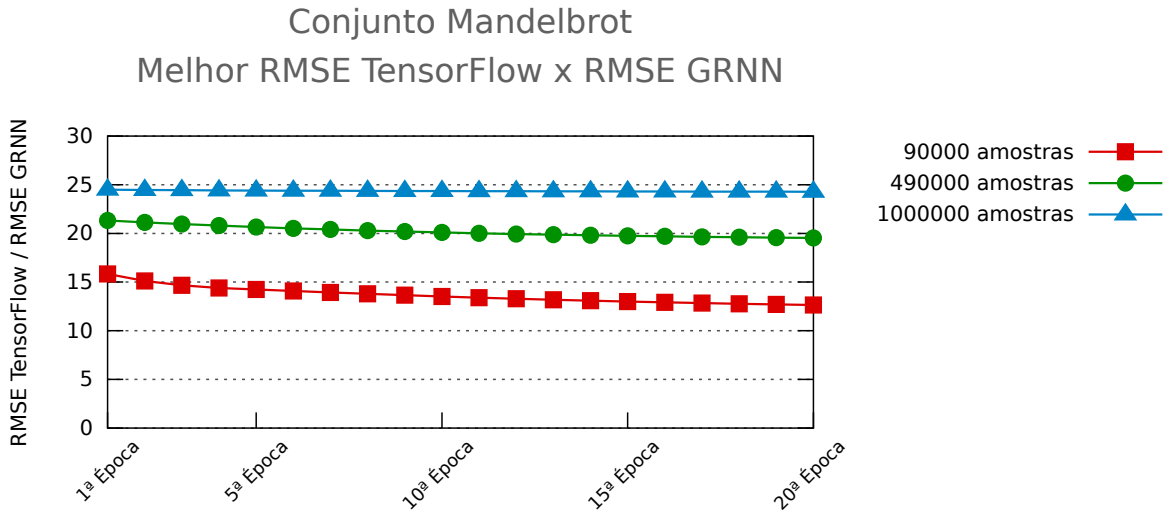

Figura 2.2: Comparativo com o melhor resultado obtido pelo estimador do TensorFlow para o problema do fractal Mandelbrot. Valor maior que 1 indica um erro menor para a GRNN e menor que 1 indica um erro menor para o TensorFlow. O erro apresentado pela GRNN foi melhor em todos os casos.

A Figura 2.2 apresenta o quociente de comparação com o melhor resultado obtido pelo TensorFlow dos dez treinamentos que foram realizados para cada conjunto de treinamento.

A Figura 2.3 apresenta o quociente de comparação com a média dos resultados obtidos pelo TensorFlow dos dez treinamentos que foram realizados, que ficou muito próxima do desempenho do melhor resultado do TensorFlow.

Diferentes estruturas da rede neural do TensorFlow foram avaliadas. Os resultados mostrados referem-se àquela que obteve os melhores resultados: uma rede neural com apenas uma camada interna com duas unidades. Para a GRNN, foi utilizado um parâmetro $\sigma = 0,01$, definido após algumas avaliações com parâmetros de diferentes tamanhos. O RMSE para cada conjunto de treinamento foi, respectivamente: $0,02$, $0,017$ e $0,015$. Dada a grandeza da variável sendo estimada, são erros expressivos e que confirmam a dificuldade do problema. Porém, são erros bem menores do que os apresentados pelo TensorFlow na obtenção de soluções para o mesmo problema.

2.3 Equação do Calor

As amostras de soluções para a equação do calor foram geradas por um algoritmo de diferenças finitas, implementado com o método *Crank-Nicolson*. Como demonstrado no Apêndice B, essa escolha garante a estabilidade das soluções geradas e, consequentemente, a consistência do conjunto de treinamento.

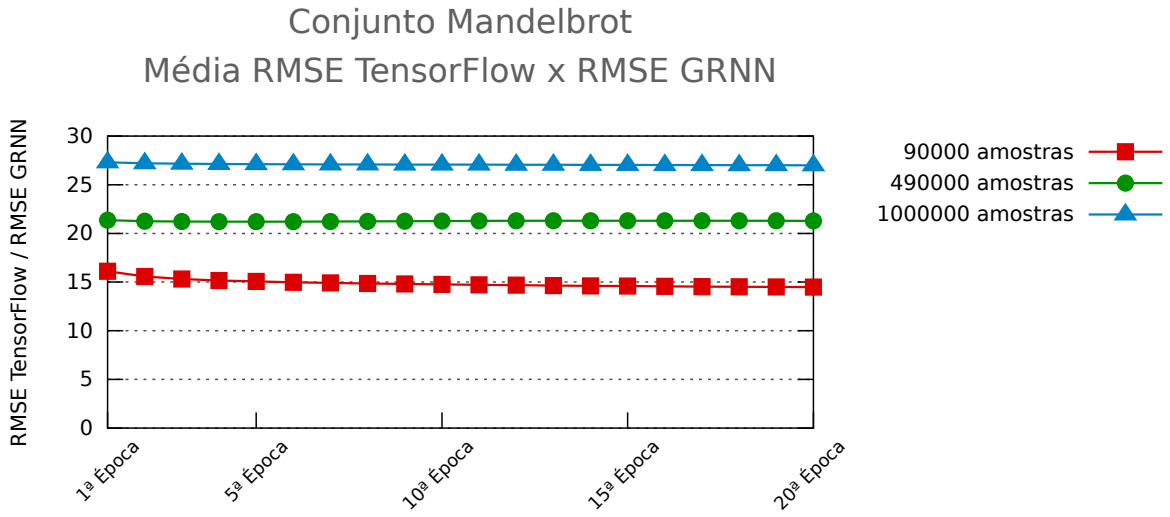

Figura 2.3: Comparativo com a média dos resultados obtidos pelo estimador do TensorFlow para o problema do fractal Mandelbrot. Valor maior que 1 indica um erro menor para a GRNN e menor que 1 indica um erro menor para o TensorFlow. O erro apresentado pela GRNN foi melhor em todos os casos.

O vetor do estado inicial $\mathbf{w}^t$ é a variável independente recebida pelo estimador e o vetor do estado posterior $\mathbf{w}^{t+1}$ é a variável dependente que será estimada. O número de dimensões dessas variáveis corresponde às dimensões dos vetores. As soluções conhecidas associadas às respectivas condições iniciais compõem o conjunto de treinamento utilizado pela GRNN.

Todos os intervalos foram normalizados para o comprimento unitário. Ou seja, tanto os valores nas componentes da variável independente quanto os valores na variável dependente vão de zero a um. Também o comprimento assumido para o material hipotético sendo simulado vai de zero a um e a constante escolhida para equação do calor é um. Nessas condições, os valores obtidos no material hipotético tendem a se estabilizar rapidamente (em poucos segundos) em torno do valor estabelecido na extremidade com o critério de Dirichlet, por isso o intervalo de tempo usado para obtenção das amostras foi $\frac{1}{10}$ de segundo.

Os valores da condição inicial foram gerados aleatoriamente, no modelo *random walk*, sem utilizar saltos muito grandes para ficar próximo do que se observa na realidade (o valor em cada ponto da curva nunca varia mais que $\frac{1}{4}$ em relação ao ponto anterior). A Figura 2.4 é um exemplo de uma curva inicial e uma curva de solução produzida pelo método de diferenças finitas.

Diferente do problema de estimar o fractal Mandelbrot, a estimativa da solução para a equação do calor permite avaliar o desempenho da GRNN para variáveis com dimensões mais altas. No caso dos testes aplicados para o problema de valor inicial da equação do

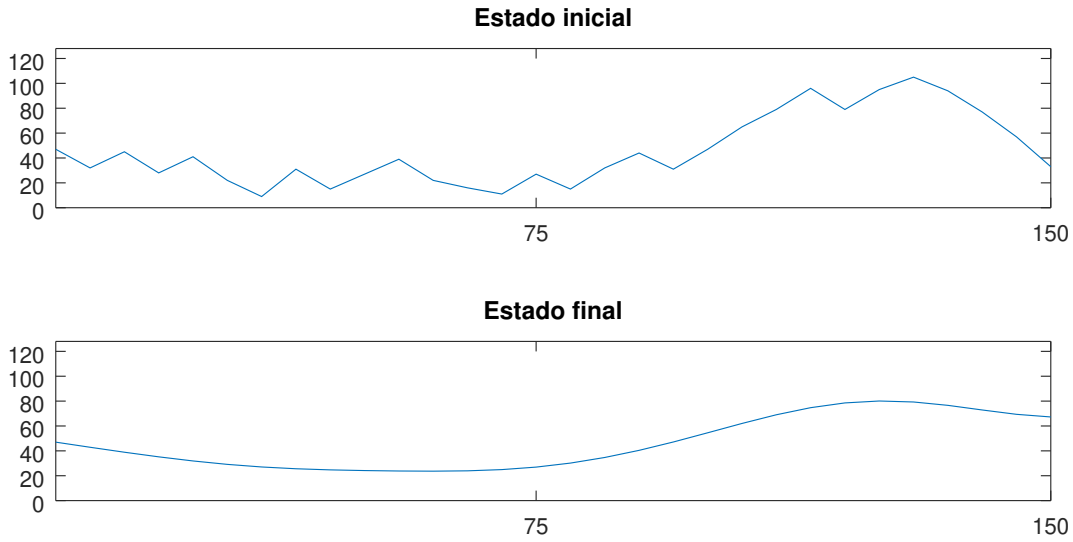

Figura 2.4: Representação gráfica de um par amostral produzido pelo método de diferenças finitas.

calor, as dimensões da variável independente e dependente foram fixadas em quatro. Ou seja, o vetor com os valores do estado inicial contém quatro valores e o vetor do estado futuro também contém quatro valores.

Foram gerados quatro conjuntos de treinamento para serem utilizados na comparação, cada um com 2^{17}, 2^{18}, 2^{19} e 2^{20} pares de variáveis independentes e dependentes. O conjunto de teste conta com 2^{12} (8192) amostras e foi o mesmo utilizado para validar os diferentes conjuntos de treinamento.

Dada a natureza linear da relação entre a variável independente e a variável dependente na solução multidimensional da equação do calor, a rede neural do TensorFlow para este problema foi construída com apenas uma camada interna, com os mesmo quatro componentes que correspondem às dimensões das variáveis. Essa estrutura se mostrou mais precisa do que as redes com um maior número de camadas e elementos internos.

Os melhores resultados para a rede neural com TensorFlow que estima a solução da equação do calor foram obtidos após o treinamento em lotes de 16 amostras com 512 passos, repetidos em 20 épocas. Dada a natureza aleatória do estado da rede neural no início do treinamento, uma mesma rede neural do TensorFlow geralmente não obtém o mesmo resultado quando treinada novamente, mesmo utilizado parâmetros idênticos. Por isso, cada treinamento foi repetido 10 vezes para se obter um panorama de resultados esperados. Para a GRNN, foi utilizado um parâmetro $\sigma = 0,2$, definido após algumas avaliações com parâmetros de diferentes tamanhos. O RMSE para cada conjunto de treinamento foi, respectivamente: $0,0057$, $0,0046$, $0,0038$ e $0,0032$. Como esperado, as estimativas para as soluções da equação do calor foram mais precisas do que as estimativas para o conjunto Mandelbrot, mesmo se tratando de variáveis com maior número de dimensões.

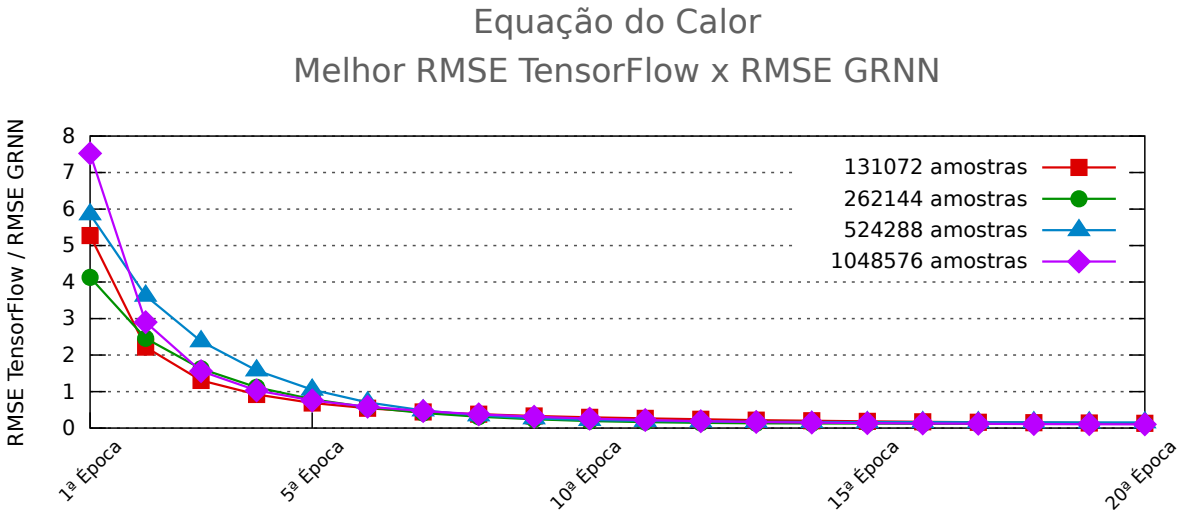

Figura 2.5: Comparativo com o melhor resultado obtido pelo estimador do TensorFlow para o problema da equação do calor. Valor maior que 1 indica um erro menor para a GRNN e menor que 1 indica um erro menor para o TensorFlow. O erro apresentado pela GRNN já foi pior a partir da quinta época de treinamento.

2.3.1 Erro Comparado

Novamente, o erro apresentado pela GRNN foi utilizado como denominador em uma quociente cujo numerador é o erro apresentado pelo estimador do TensorFlow. Dessa forma, o valor do quociente será superior a 1 se o erro no TensorFlow for superior ao erro da GRNN e inferior a 1 se o erro no TensorFlow for inferior ao erro da GRNN.

A Figura 2.5 apresenta o quociente de comparação com o melhor resultado obtido pelo TensorFlow dos dez treinamentos que foram realizados. Nesse gráfico é possível observar que o TensorFlow obteve um erro menor para o mesmo conjunto de validação já na quarta e quinta épocas de treinamento.

Porém, quando considerada a média dos dez treinamentos que foram realizados, os quocientes de comparação apresentam resultados bastante diferentes. Como demonstrado na Figura 2.6, o erro apresentado pelo estimador do TensorFlow em 20 épocas não foi melhor que o erro da GRNN nos quatro maiores conjuntos.

Diante desses resultados, pode-se afirmar que a GRNN oferece vantagem quando se deseja obter estimativas mais imediatas, quando não há tempo para realizar várias épocas de treinamento no TensorFlow. Após o treinamento, no entanto, o TensorFlow oferece estimativas mais precisas. Por isso, a GRNN será preferível quando o conjunto de treinamento é volátil e a estimativa precisa ser produzida rapidamente.

Apesar de não ter sido objeto de análise minuciosa nos testes comparativos, vale ressaltar a grande discrepância no tempo utilizado por cada método para produzir as estimativas. O treinamento da rede neural do TensorFlow pode demorar muitos minutos

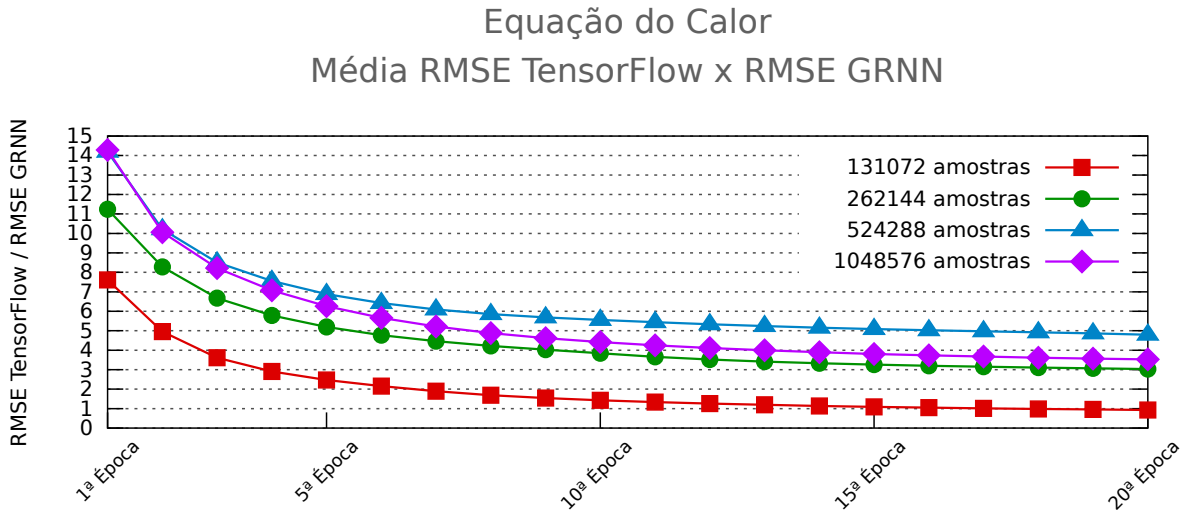

Figura 2.6: Comparativo com a média dos resultados obtidos pelo estimador do TensorFlow para o problema da equação do calor. Valor maior que 1 indica um erro menor para a GRNN e menor que 1 indica um erro menor para o TensorFlow. No teste, O TensorFlow superou a GRNN apenas com o conjunto de treinamento com 131072 amostras.

ou até horas, mesmo quando executado em GPU. Como pode ser verificado no Capítulo 3, o tempo necessário para produzir uma estimativa com a GRNN paralelizada em GPU dificilmente ultrapassa 1 segundo, mesmo quando utilizando hardware de baixo custo e um conjunto de treinamento com centenas de milhares de amostras.

Capítulo 3

Desempenho da Paralelização

A partir das premissas delineadas no Capítulo 1, foi implementada uma versão paralela do algoritmo GRNN em GPU. O ganho de desempenho foi avaliado em relação à implementação sequencial da GRNN. Ambas implementações produzem exatamente as mesmas estimativas, pois constroem a superfície de regressão a partir dos mesmos conjuntos de treinamento.

Além da paralelização em GPU, também foi implementada uma versão paralela da GRNN em CPU utilizando a biblioteca *pthreads*. Conforme poderá ser verificado nos resultados obtidos, cada modalidade de paralelização tem suas particularidades e uma implementação ingênua pode não obter ganhos significativos de desempenho.

A versão sequencial e paralelizada em CPU foi testada em dois modelos de CPU, descritos na tabela 3.1.

Modelo	**Clock**	**Núcleos**	**Processadores**
Intel Core i7-4770	3,4GHz	4	8
Intel Xeon E5-2690 v2	3,0GHz	20	40

Tabela 3.1: Modelos de CPU utilizadas nos testes

A versão paralelizada em GPU foi testada em três diferentes modelos de GPU, descritas na tabela 3.2.

Modelo	**Arquitetura**	**CUDA Cores**
GeForce GTX 750 Ti	Maxwell	640
Tesla K40c	Kepler	2880
GeForce GTX TITAN X	Maxwell	3072

Tabela 3.2: Modelos de GPU utilizadas nos testes

Todos os testes de desempenho se basearam nos mesmos problemas e conjuntos de treinamento elaborados no Capítulo 2. Foi contabilizado o tempo para realizar 8192 estimativas para cada um dos conjuntos de treinamento. As linhas dos gráficos corres-

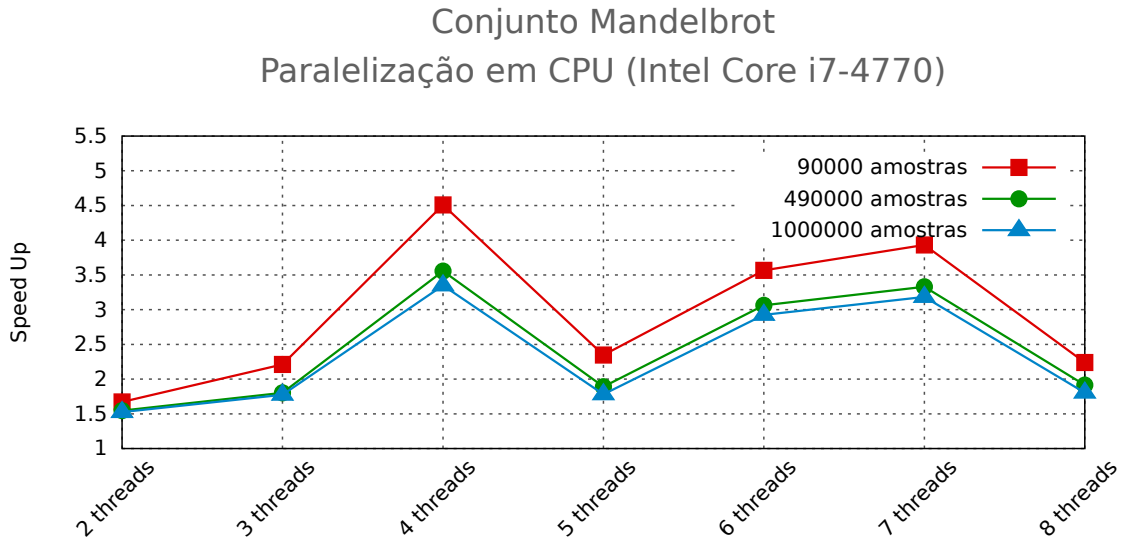

Figura 3.1: Desempenho da paralelização em CPU com oito núcleos ao estimar um ponto no conjunto Mandelbrot. O gráfico mostra o ganho em relação a execução sequencial ao paralelizar em 2, 3, 4, 5, 6, 7 e 8 threads.

pondem ao ganho (*Speed Up*) em relação à versão sequencial. O cálculo foi feito usando o tempo médio gasto em 10 execuções para cada caso.

No caso dos testes em GPU, foi considerado apenas o tempo de processamento e desconsiderado o tempo despendido para copiar os dados do computador anfitrião para a memória da GPU, pois este tempo é fortemente influenciado pela velocidade de comunicação do barramento PCI. Em geral, esse tempo de cópia correspondeu a menos de 0,1% do total de tempo gasto na estimativa, por isso foi considerado desprezível e a atenção voltada apenas para o processamento na GPU.

3.1 Paralelização em CPU

Os gráficos nas Figuras 3.1, 3.2, 3.3 e 3.4 mostram o ganho de desempenho da GRNN paralelizada em CPU. O simples particionamento do conjunto de treinamento e sua distribuição nas unidades de processamento disponíveis não foi suficiente para obter o ganho linear esperado. Além disso, é possível notar que o ganho em relação à versão sequencial diminui conforme aumenta o tamanho no conjunto de treinamento.

O comportamento errante do desempenho em CPU indica o quão influente é a arquitetura do hardware empregado na implementação. A simples paralelização do algoritmo, sem levar em conta possíveis gargalos no acesso à memória, pode até deteriorar o desempenho em relação a versão sequencial.

Além disso, cada tipo de problema pode ter uma predileção por um tipo de arquitetura. No caso da GRNN, o grande poder computacional oferecido por cada unidade de

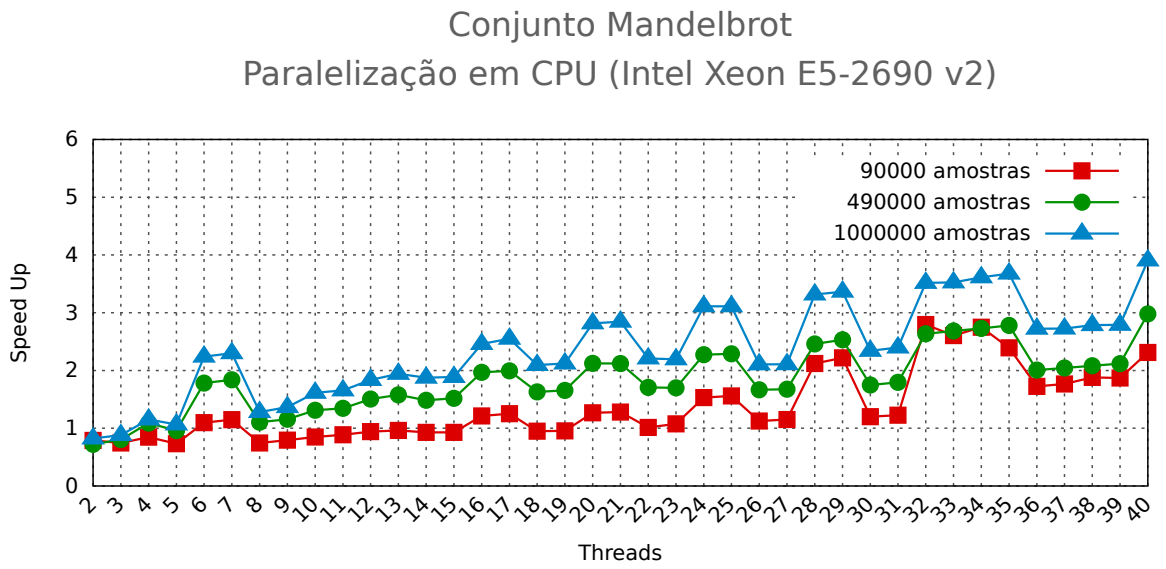

Figura 3.2: Desempenho da paralelização em CPU com 40 núcleos ao estimar um ponto no conjunto Mandelbrot. O gráfico mostra o ganho em relação a execução sequencial.

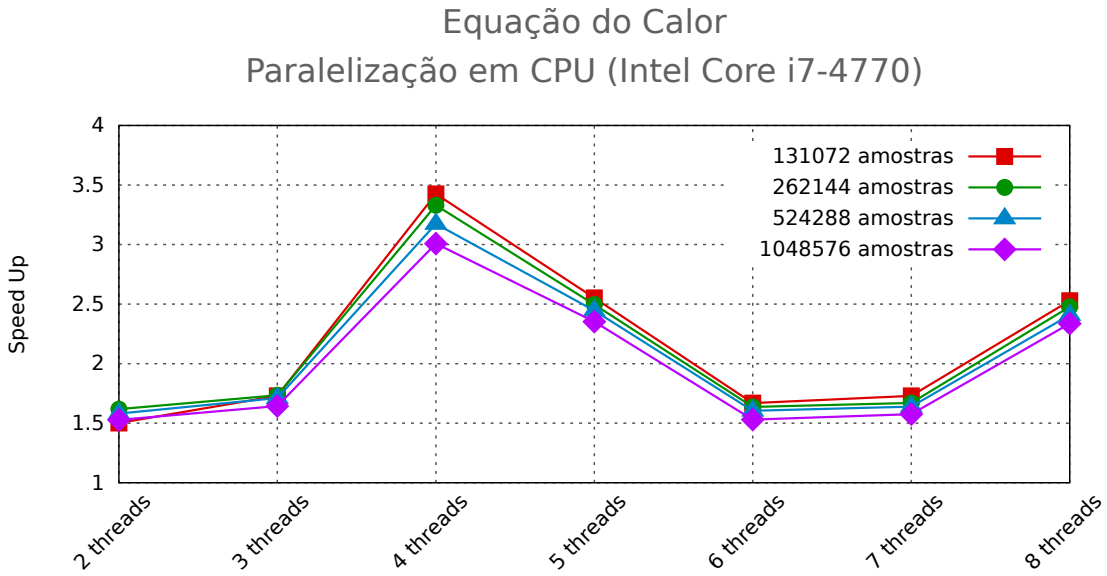

Figura 3.3: Desempenho médio das estimativas para solução da equação do calor. Ganho no tempo de execução em CPU ao paralelizar em 2, 3, 4, 5, 6, 7 e 8 threads.

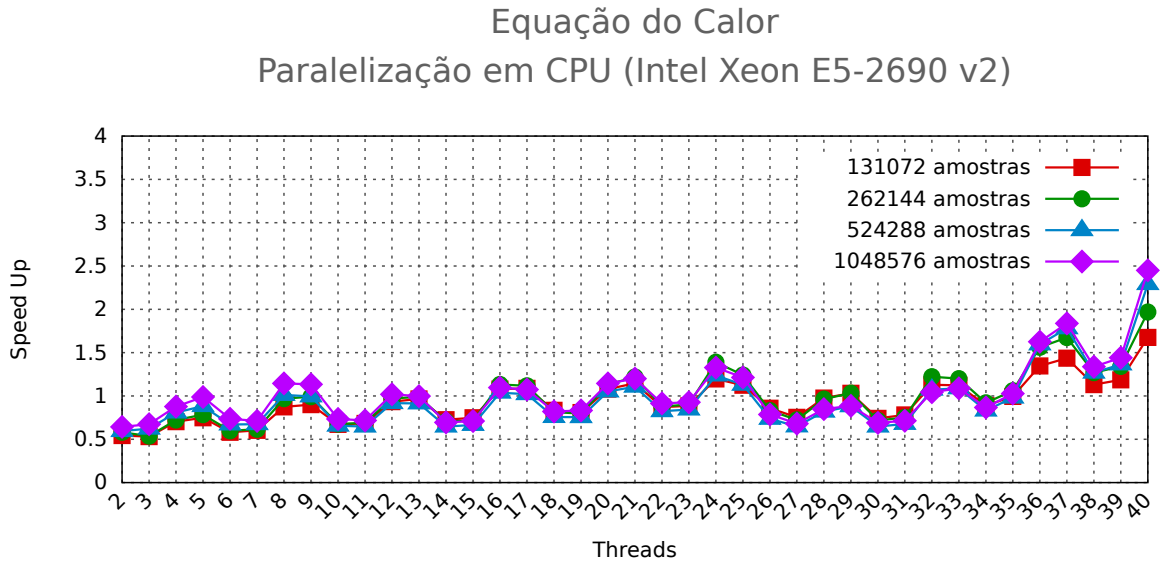

Figura 3.4: Estimativa para solução da equação do calor. Ganho no tempo de execução em CPU com 40 núcleos.

processamento nas CPUs parece não fazer diferença para realizar as operações individualmente pouco intensivas exigidas pelo algoritmo.

O modelo *Single Instruction, Multiple Data* adotado nas GPUs é mais apropriado para esse tipo de algoritmo, onde a mesma operação simples precisa ser repetida num grande volume de dados.

3.2 Paralelização em GPU

Os mesmos conjuntos de dados foram utilizados para avaliar o desempenho da versão paralela da GRNN em GPU e também foi computada a média de tempo em dez execuções. A base de comparação foi a execução da versão sequencial na CPU Intel Core i7-4770. Os tempos de execução absolutos variaram bastante dependendo do modelo de GPU utilizado. Por exemplo, o tempo médio que a GeForce GTX TITAN X leva para estimar as 8192 soluções para a equação do calor usando o conjunto com 2^{20} amostras é aproximadamente $2,1$ segundos, enquanto que a Tesla K40c e a GeForce GTX 750 Ti demoram aproximadamente $3,5$ e $8,5$ segundos para executar a mesma tarefa. Mesmo no pior dos casos, portanto, é necessário apenas $0,001$ segundo para computar a estimativa usando um conjunto de treinamento com 1.048.576 amostras.

Os gráficos nas Figuras 3.5 e 3.6 comparam o desempenho da GRNN sequencial em CPU com a paralelização em GPU para os dois problemas abordados. O ganho no tempo de execução foi bastante nítido nos três modelos de GPU testados e é ainda melhor quando a variável dependente é multidimensional, como é o caso da solução para a equação do calor.

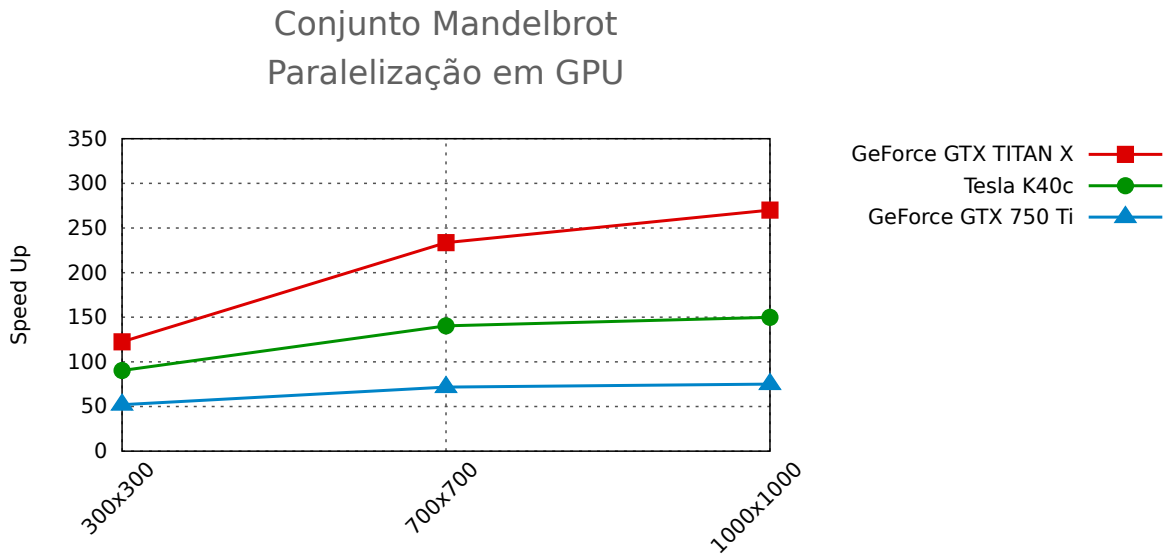

Figura 3.5: Desempenho médio das estimativas para um ponto no conjunto Mandelbrot. O gráfico apresenta o ganho em desempenho da GRNN em três diferentes modelos de GPU em relação à versão sequencial da GRNN em CPU.

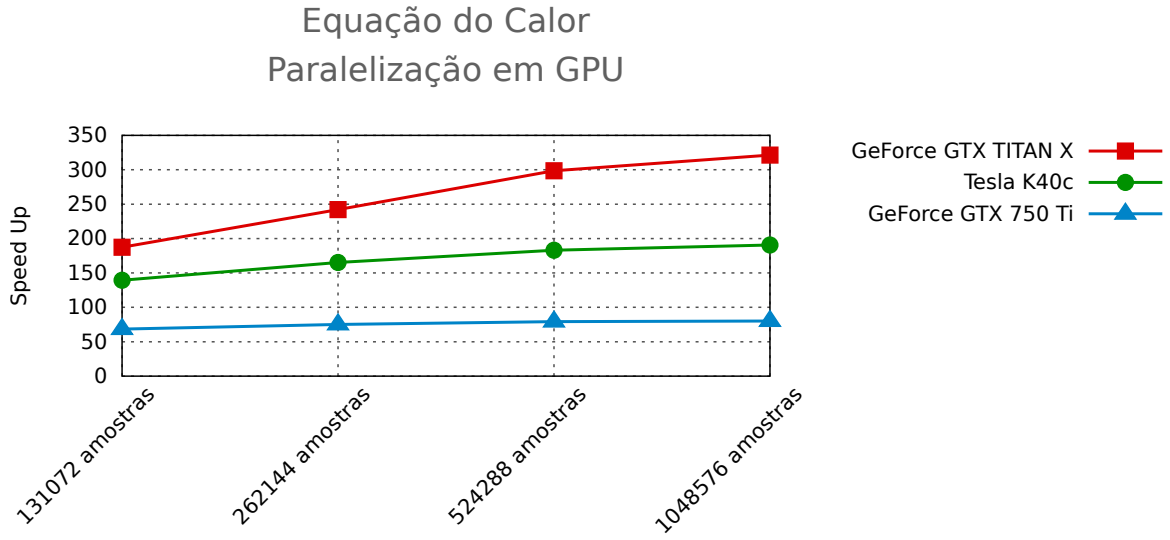

Figura 3.6: Estimativa para solução da equação do calor. O gráfico apresenta o ganho em desempenho da GRNN em três diferentes modelos de GPU em relação à versão sequencial da GRNN em CPU.

Os resultados são ainda mais interessantes se levado em conta o custo dos equipamentos empregados. Os melhores resultados obtidos nas implementações em CPU são superados em mais de dez vezes pelos piores resultados obtidos na paralelização em GPU, produzidos pela já defasada Geforce GTX 750 Ti, cujo custo é uma pequena fração do custo das CPUs utilizadas.

Além disso, é possível notar a tendência de ganho no desempenho conforme aumenta o tamanho do conjunto de treinamento, o que sugere o potencial de escalabilidade da GRNN em GPU.

Capítulo 4

Conclusão

Os resultados obtidos mostram que a GRNN oferece benefícios quando se deseja obter estimativas mais imediatas, quando não há tempo para realizar várias épocas de treinamento com uma ferramenta como o TensorFlow. A GRNN será preferível se o conjunto de treinamento for volátil e a estimativa precisa ser produzida rapidamente. Já um treinamento de rede neural profunda pode demorar muitos minutos ou até horas, mesmo quando executado em GPU.

O tempo necessário para produzir uma estimativa com a GRNN paralelizada em GPU dificilmente ultrapassa 1 segundo, mesmo quando utilizando hardware de baixo custo e um conjunto de treinamento com centenas de milhares de amostras.

O comportamento errante do desempenho em CPU indica o quão influente é a arquitetura do hardware empregado na implementação. A simples paralelização do algoritmo, sem levar em conta possíveis gargalos no acesso à memória, pode até deteriorar o desempenho em relação a versão sequencial. No caso da paralelização da GRNN em CPU, o grande poder computacional oferecido por cada unidade de processamento não ofereceu ganhos.

Já os ganhos em desempenho obtidos pela GRNN paralelizada em GPU foram bastante expressivos. Os melhores resultados obtidos nas implementações em CPU foram superados em mais de dez vezes pelos piores resultados obtidos na paralelização em GPU em hardware já defasado, cujo custo é uma pequena fração do custo das CPUs utilizadas. Quando são utilizadas GPUs atuais, os ganhos em performance superam em centenas de vezes as implementações em CPU.

Apêndice A

Estimador de Densidade de Probabilidade

Seja X uma variável aleatória real contínua cuja densidade de probabilidade é dada pela função $f(x)$. Por definição:

$$f(x) \geq 0 \quad , \quad \int_{-\infty}^{+\infty} f(x)\, dx = 1$$

e a probabilidade de $x \sim X$ estar no intervalo $(z-h, z+h)$ para $z, h \in \mathbb{R}$ fixos é dada por

$$\mathbb{P}(z-h < X < z+h) = F(x)|_{z-h}^{z+h} = \int_{z-h}^{z+h} f(x)\, dx$$

onde a função $F(x)$ é chamada distribuição de probabilidade de X.

Quando desconhecida, a densidade de X pode ser estimada a partir de um número finito de amostras $x^{(i)} \sim X$, $i = 1, 2, \cdots, n$. Para isso, é adotado um valor h "pequeno" aplicado a uma aproximação da densidade dada por

$$\hat{f}(x) \approx \frac{F(x+h) - F(x-h)}{2h} \tag{A.1}$$

onde a expressão $F(x+h) - F(x-h)$ é substituída pela razão entre a quantidade de amostras presentes no intervalo $(x-h, x+h)$ e o número total de amostras n. Utilizando uma função de detecção amostral na forma

$$w(r) = \begin{cases} 1/2 & \text{se } |r| < 1 \\ 0 & \text{se } |r| \geq 1 \end{cases}$$

é possível obter a quantidade de amostras presentes no intervalo $(x-h, x+h)$, quantidade essa dada pela expressão

$$2\sum_{i=1}^{n} w\left(\frac{x-x^{(i)}}{h}\right). \tag{A.2}$$

Substituindo a expressão $F(x+h) - F(x-h)$ em (A.1) por (A.2), a função $\hat{f}(x)$ pode ser reescrita na forma

$$\hat{f}(x) \approx \frac{1}{nh}\sum_{i=1}^{n} w\left(\frac{x-x^{(i)}}{h}\right). \tag{A.3}$$

A aproximação definida em (A.3) é chamada *estimador ingênuo* de $f(x)$, haja vista que não é uma aproximação contínua da densidade de probabilidade. Outros estimadores são obtidos ao substituir $w\left(\frac{x-x^{(i)}}{h}\right)$ em (A.3) por qualquer outra função $k\left(\frac{x-x^{(i)}}{h}\right)$, desde que

$$k(r) \geq 0 \quad \text{e} \quad \int_{-\infty}^{+\infty} k(r)\, dr = 1$$

de modo a preservar a função $\hat{f}(x)$ como densidade de probabilidade. Em particular, se $k(x)$ for limitada e contínua em todo domínio, é possível demonstrar que $\hat{f}(x)$ permanece como densidade de probabilidade aplicando uma simples integração com troca de variável:

$$\int_{-\infty}^{+\infty} \hat{f}(t)\, dt = \int_{-\infty}^{+\infty} \frac{1}{nh}\sum_{i=1}^{n} k\left(\frac{t-x_i}{h}\right)\, dt = \frac{1}{n}\int_{-\infty}^{+\infty}\sum_{i=1}^{n} k\left(r\right)\, dr$$

$$\int_{-\infty}^{+\infty} k(r)\, dr = 1 \implies \frac{1}{n}\int_{-\infty}^{+\infty}\sum_{i=1}^{n} k\left(r\right)\, dr = \frac{1}{n}\sum_{i=1}^{n}\int_{-\infty}^{+\infty} k\left(r\right)\, dr = 1$$

$$\therefore \hat{f}(x) \approx \frac{1}{nh}\sum_{i=1}^{n} k\left(\frac{x-x^{(i)}}{h}\right). \tag{A.4}$$

A função $k(x)$ é chamada *kernel* do estimador $\hat{f}(x)$. Se $k(x)$ for contínua em todo ponto do domínio, então $\hat{f}(x)$ também será contínua no domínio.

A.1 Estimador da densidade conjunta

Sejam X e Y duas variáveis aleatórias reais contínuas, X independente e Y dependente de X. Dado um número finito de amostras $(x^{(i)}, y^{(i)}) \sim (X, Y)$, $i = 1, 2, \cdots, n$, a densidade conjunta de (X, Y) pode ser estimada por

$$\hat{f}(x,y) \approx \frac{1}{nh^2}\sum_{i=1}^{n} k\left(\frac{x-x^{(i)}}{h}\right) k\left(\frac{y-y^{(i)}}{h}\right) \tag{A.5}$$

e a função distribuição de probabilidade é o volume definido pela integral dupla em x e y:

$$\mathbb{P}(y-h<Y<y+h|x-h<X<x+h)=\int_{y-h}^{y+h}\int_{x-h}^{x+h}\hat{f}(x,y)\,dx\,dy. \tag{A.6}$$

Como no caso da distribuição de probabilidade em variável única, a integração da distribuição conjunta em todo o domínio é igual a 1.

A.2 Kernel Gaussiano

Uma escolha natural para a função Kernel é utilizar uma curva normal dada pela função Gaussiana

$$g(x)=\frac{1}{\sigma\sqrt{2\pi}}\exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right) \tag{A.7}$$

onde a média μ é definida como a i-ésima amostra $x^{(i)}$ e o desvio padrão σ é definido como o intervalo h. Adotando o Kernel

$$k(t(x))=\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{t^2(x)}{2}\right) \tag{A.8}$$

e

$$t(x)=\frac{x-x^{(i)}}{\sigma} \tag{A.9}$$

como seu parâmetro, a expressão em (A.7) é convertida para a forma

$$\frac{1}{\sigma}k\left(\frac{x-x^{(i)}}{\sigma}\right)$$

que aplicada em (A.4) resulta no estimador gaussiano

$$\hat{f}(x)\approx\frac{1}{n\sigma}\sum_{i=1}^{n}k\left(\frac{x-x^{(i)}}{\sigma}\right). \tag{A.10}$$

Diferente do estimador ingênuo, o estimador gaussiano regride para uma superfície contínua e converge assintoticamente para a distribuição adjacente em todo o domínio.

A.3 Densidade Conjunta com Kernel Gaussiano

O kernel gaussiano aplicado à estimativa da densidade conjunta (A.5) resulta na aproximação

$$\hat{f}(x,y) \approx \frac{1}{n\sigma^2\sqrt{2\pi}} \sum_{i=1}^{n} \exp\left(-\frac{1}{2}\left(\frac{x-x^{(i)}}{\sigma}\right)^2\right) \exp\left(-\frac{1}{2}\left(\frac{y-y^{(i)}}{\sigma}\right)^2\right). \tag{A.11}$$

Como demonstrou Cacoullos [3], a aproximação pode ser extrapolada para X multivariada. Nesse caso, a aproximação em (A.11) assume a forma

$$\hat{f}(\mathbf{x},y) \approx \frac{1}{n\sigma^{p+1}(2\pi)^{\frac{p+1}{2}}} \sum_{i=1}^{n} \exp\left(-\frac{1}{2}\left(\frac{d}{\sigma}\right)^2\right) \exp\left(-\frac{1}{2}\left(\frac{y-y^{(i)}}{\sigma}\right)^2\right) \tag{A.12}$$

onde p é o número de dimensões do vetor $\mathbf{x}$ e d é uma medida de distância entre x e $\mathbf{x}^{(i)}$. Em princípio, qualquer conceito de distância do espaço vetorial pode ser utilizado. Neste trabalho, foi utilizada o quadrado da distância euclidiana, $d = (\mathbf{x}-\mathbf{x}^{(i)})^T(\mathbf{x}-\mathbf{x}^{(i)})$.

A.4 Esperança da Variável Dependente

Para estimar a variável dependente, é necessário determinar a distribuição marginal da distribuição conjunta dada a variável independente, ou seja, a distribuição marginal em $\mathbf{X} = \mathbf{x}$. Essa distribuição marginal é dada pela integral

$$f_X(x) = \int_{-\infty}^{+\infty} \hat{f}(x,y)\, dy \tag{A.13}$$

e a partir dela é construída a distribuição de probabilidade da variável dependente Y dada $\mathbf{X} = \mathbf{x}$:

$$\mathbb{P}(z-h < Y < z+h | \mathbf{X} = \mathbf{x}) = \frac{\int_{z-h}^{z+h} \hat{f}(x,y)\, dy}{f_X(x)}. \tag{A.14}$$

Decorre de (A.14) que a densidade da variável dependente, chamada $\bar{f}(y)$, está definida por

$$\bar{f}(y) := \frac{\hat{f}(x,y)}{f_X(x)}. \tag{A.15}$$

Logo, a esperança de Y dada $\mathbf{X} = \mathbf{x}$ fica definida por

$$\mathbb{E}(Y|\mathbf{X} = \mathbf{x}) = \int_{-\infty}^{+\infty} y\bar{f}(y)dy = \frac{\int_{-\infty}^{+\infty} y\hat{f}(x,y)\, dy}{\int_{-\infty}^{+\infty} \hat{f}(x,y)\, dy}. \tag{A.16}$$

Expandindo a função $\hat{f}(x,y)$ encontrada em (A.16), chega-se às somatórias para o numerador e denominador do quociente:

$$\mathbb{E}(Y|\mathbf{X}=\mathbf{x}) = \frac{\sum_{i=1}^{n} \exp\left(-\frac{1}{2}\left(\frac{d}{\sigma}\right)^2\right) \int_{-\infty}^{+\infty} y \exp\left(-\frac{1}{2}\left(\frac{y-y^{(i)}}{\sigma}\right)^2\right) dy}{\sum_{i=1}^{n} \exp\left(-\frac{1}{2}\left(\frac{d}{\sigma}\right)^2\right) \int_{-\infty}^{+\infty} \exp\left(-\frac{1}{2}\left(\frac{y-y^{(i)}}{\sigma}\right)^2\right) dy} \tag{A.17}$$

que contém duas integrais cujas soluções são dadas por

$$\int_{-\infty}^{+\infty} y \exp\left(-\frac{1}{2}\left(\frac{y-y^{(i)}}{\sigma}\right)^2\right) dy = y^{(i)}\sqrt{2\pi\sigma^2} \tag{A.18}$$

e

$$\int_{-\infty}^{+\infty} \exp\left(-\frac{1}{2}\left(\frac{y-y^{(i)}}{\sigma}\right)^2\right) dy = \sqrt{2\pi\sigma^2}. \tag{A.19}$$

Finalmente, as expressões (A.18) e (A.19) são utilizadas em (A.17) e chega-se à expressão

$$\mathbb{E}(Y|\mathbf{X}=\mathbf{x}) = \frac{\sum_{i=1}^{n} y^{(i)} \exp\left(-\frac{1}{2}\left(\frac{d}{\sigma}\right)^2\right)}{\sum_{i=1}^{n} \exp\left(-\frac{1}{2}\left(\frac{d}{\sigma}\right)^2\right)} \tag{A.20}$$

Apêndice B

Método de Diferenças Finitas

A equação diferencial parcial parabólica

$$u_t = \alpha u_{xx} \tag{B.1}$$

definida em $0 \leq x \leq L$, $0 \leq t$ e α constante positiva é conhecida como *equação do calor* e é amplamente utilizada para modelar e prever fenômenos físicos relacionados à difusão de energia num meio ao longo do tempo.

Uma função $u(x,t)$ é dita solução da equação do calor se satisfaz a igualdade definida em (B.1). Se forem conhecidos os valores de $u(x,t)$ para um instante inicial $t = 0$ — por convenção associados a uma função $\Phi(x)$ — será possível aferir o valor de $u(x,t)$ num instante posterior a $t = 0$. Os valores da função no instante inicial são utilizados para obter aproximações da derivada parcial de segunda ordem u_{xx}, que por (B.1) está relacionada à derivada parcial u_t, utilizada para estimar a solução num instante adiante no tempo.

O primeiro passo do método de Diferenças Finitas é a adoção de um modelo discreto para as soluções. São escolhidos m pontos igualmente espaçados ao longo da condição inicial $\Phi(x)$, ou seja, é definido um vetor $\mathbf{w} \in \mathbb{R}^m$ tal que

$$\begin{matrix} \mathbf{w}_i = \Phi(ih) \\ 0 \leq i < m \end{matrix} \begin{cases} \Phi : X \subset \mathbb{R} \to \mathbb{R}, X = \{x \in \mathbb{R} | 0 \leq x \leq L\} \\ h = \frac{L}{m-1}, m \in \mathbb{N}, m > 2 \end{cases}$$

O objetivo é utilizar um vetor $\mathbf{w}$ cujos valores são conhecidos no instante t — denominado $\mathbf{w}^{(t)}$ — para gerar os valores num vetor no instante adiante, denominado $\mathbf{w}^{(t+1)}$.

A notação na forma $\mathbf{w}_{i,j}$ indica a aproximação de $u(ih, jk)$, onde k é um tamanho de passo conveniente no tempo e $j \in \mathbb{N}$.

Subentende-se por (B.1) que $u(x,t)$ possui as derivadas parciais em todo domínio e os valores $\mathbf{w}_{i,j}$ estão igualmente espaçados em x e t (haja vista que os valores h e k são constantes). Assim, é possível fazer uma aproximação numérica da derivada de segunda

ordem

$$u_{xx}(ih, jk) \approx \frac{\mathbf{w}_{i-1,j} - 2\mathbf{w}_{i,j} + \mathbf{w}_{i+1,j}}{h^2} \tag{B.2}$$

para $0 < i < m - 1$. Essa aproximação é conhecida como *diferença centrada* em i e converge para $u_{xx}(ih, jk)$ conforme $h \to 0$.

Já a derivada de primeira ordem pode ser aproximada por

$$u_t(ih, jk) \approx \frac{\mathbf{w}_{i,j+1} - \mathbf{w}_{i,j}}{k} \tag{B.3}$$

e, como em (B.2), converge para $u_t(ih, jk)$ conforme $k \to 0$.

Aplicando as expressões (B.2) e (B.3), se obtêm duas formulações válidas para (B.1). A primeira é conhecida como *diferença adiante*:

$$\frac{\mathbf{w}_{i,j+1} - \mathbf{w}_{i,j}}{k} = \alpha \left(\frac{\mathbf{w}_{i-1,j} - 2\mathbf{w}_{i,j} + \mathbf{w}_{i+1,j}}{h^2} \right) \tag{B.4}$$

A segunda é conhecida como *diferença atrás*:

$$\frac{\mathbf{w}_{i,j+1} - \mathbf{w}_{i,j}}{k} = \alpha \left(\frac{\mathbf{w}_{i-1,j+1} - 2\mathbf{w}_{i,j+1} + \mathbf{w}_{i+1,j+1}}{h^2} \right) \tag{B.5}$$

A forma em (B.4) pode apresentar problemas de estabilidade a depender do tamanho dos passos escolhidos, o que não acontece em (B.5). Aproximações estáveis [14] que convergem mais rapidamente são obtidas pelo método *Crank-Nicolson*, formulado a partir da média entre (B.4) e (B.5). Adotando a notação $\mathbf{w}^{(t)}$ e $\mathbf{w}^{(t+1)}$:

$$\frac{\mathbf{w}_i^{(t+1)} - \mathbf{w}_i^{(t)}}{k} = \frac{\alpha}{2} \left(\frac{\mathbf{w}_{i-1}^{(t)} - 2\mathbf{w}_i^{(t)} + \mathbf{w}_{i+1}^{(t)}}{h^2} + \frac{\mathbf{w}_{i-1}^{(t+1)} - 2\mathbf{w}_i^{(t+1)} + \mathbf{w}_{i+1}^{(t+1)}}{h^2} \right) \tag{B.6}$$

Tomando $\lambda = \frac{k\alpha}{h^2}$ e simplificando (B.6), se obtém a forma

$$-\frac{\lambda}{2}\mathbf{w}_{i-1}^{(t+1)} + (1+\lambda)\mathbf{w}_i^{(t+1)} - \frac{\lambda}{2}\mathbf{w}_{i+1}^{(t+1)} = \frac{\lambda}{2}\mathbf{w}_{i-1}^{(t)} + (1-\lambda)\mathbf{w}_i^{(t)} + \frac{\lambda}{2}\mathbf{w}_{i+1}^{(t)} \tag{B.7}$$

onde os valores correspondentes ao instante posterior estão do lado esquerdo da igualdade (B.7) e os valores correspondentes ao instante anterior estão do lado direito. Esse formato possibilita a adoção de operações matriciais para obtenção das soluções.

Sejam $A^{(i)}, B^{(i)} \in \mathbb{R}^m$ as respectivas i-ésimas linhas de duas matrizes A e B tais que:

$$A^{(i)}\mathbf{w}^{(t)} = -\frac{\lambda}{2}\mathbf{w}_{i-1}^{(t)} + (1+\lambda)\mathbf{w}_i^{(t)} - \frac{\lambda}{2}\mathbf{w}_{i+1}^{(t)} \tag{B.8}$$

$$B^{(i)}\mathbf{w}^{(t)} = \frac{\lambda}{2}\mathbf{w}_{i-1}^{(t)} + (1-\lambda)\mathbf{w}_i^{(t)} + \frac{\lambda}{2}\mathbf{w}_{i+1}^{(t)} \tag{B.9}$$

então, para $0 < i < m-1$, a forma matricial de (B.7) é dada por

$$A\mathbf{w}^{(t+1)} = B\mathbf{w}^{(t)} \tag{B.10}$$

onde $\mathbf{w}^{(t+1)}$ é o vetor dos valores no instante posterior, $\mathbf{w}^{(t)}$ é o vetor no instante anterior e as matrizes A e B são tridiagonais tais que

$$a_{ik} = \begin{cases} 1+\lambda & \text{se } i = k \\ -\frac{\lambda}{2} & \text{se } |i-k| = 1 \\ 0 & \text{se } |i-k| > 1 \end{cases}$$

$$b_{ik} = \begin{cases} 1-\lambda & \text{se } i = k \\ \frac{\lambda}{2} & \text{se } |i-k| = 1 \\ 0 & \text{se —i - k—} > 1 \end{cases}$$

Porém, essa forma matricial se aplica somente aos pontos interiores dos vetores nos instantes anterior e posterior. A forma para as fronteiras inicial e final deve ser analisada mais cuidadosamente.

B.1 Condições de fronteira

Um material homogêneo, cujo calor numa fronteira seja igual a um valor arbitrário (possivelmente determinado pela ação de um aquecedor ou resfriador) é modelado aplicando o critério de *Dirichlet*, ou seja, o valor nessa fronteira é imposto por uma função $f(t)$ potencialmente dissociada do estado do material.

Se o calor na fronteira deve se manter proporcional ao restante do material, é aplicada o critério de *Neumann*. Nela, o valor da derivada em x é definido por uma função $g(t)$. Por exemplo, o critério de Neumann com uma função $g(t) = 0$ determina que o valor na fronteira seja igual ao do ponto imediatamente ao lado (derivada nula).

A forma matricial para o critério de Dirichlet é bastante simples. Se o valor na fronteira $l \in \{0, m-1\}$ é imposto por $f(t)$, então basta substituir o valor em $\mathbf{w}_l^{(t)}$ por $f(t+1)$ e definir uma operação para copiar $\mathbf{w}_l^{(t)}$ em $\mathbf{w}_l^{(t+1)}$

$$\mathbf{w}_l^{(t+1)} = \mathbf{w}_l^{(t)} \implies A^{(l)} = B^{(l)} = \mathbf{e}^{(l)} \tag{B.11}$$

onde $\mathbf{e}^{(l)} \in \mathbb{R}^m$ é o vetor-linha com o valor 1 na l-ésima coordenada e 0 em todas as outras.

Já o critério de Neumann impõe que existe uma derivada — dada por $g(t)$ — na fronteira, que deve ser utilizada para gerar $\mathbf{w}_l^{(t+1)}$, $l \in \{0, m-1\}$. A igualdade (B.7) não pode ser utilizada para aproximar o valor na fronteira, pois contém índices inválidos de $\mathbf{w}$ quando $l = 0$ ou $l = m - 1$.

Esse impedimento pode ser contornado incluindo um pseudo-ponto extra além da fronteira. Sem perda de generalidade, decorre do critério de Neumann na fronteira em L que

$$\text{Pseudo-ponto:} \quad \mathbf{w}_m^{(t)} = \mathbf{w}_{m-1}^{(t)} + hg(t) \tag{B.12}$$

Substituindo em (B.7), se obtém

$$\begin{aligned} -\frac{\lambda}{2}\mathbf{w}_{m-2}^{(t+1)} + (1+\lambda)\mathbf{w}_{m-1}^{(t+1)} - \frac{\lambda}{2}\left(\mathbf{w}_{m-1}^{(t+1)} + hg(t+1)\right) \\ = \frac{\lambda}{2}\mathbf{w}_{m-2}^{(t)} + (1-\lambda)\mathbf{w}_{m-1}^{(t)} + \frac{\lambda}{2}\left(\mathbf{w}_{m-1}^{(t)} + hg(t)\right) \end{aligned} \tag{B.13}$$

resultando nas formas matriciais

$$A^{(m-1)}\mathbf{w}^{(t)} = -\frac{\lambda}{2}\mathbf{w}_{m-2}^{(t)} + \left(1 + \frac{\lambda}{2}\right)\mathbf{w}_{m-1}^{(t)} - \frac{\lambda}{2}hg(t) \tag{B.14}$$

$$B^{(m-1)}\mathbf{w}^{(t)} = \frac{\lambda}{2}\mathbf{w}_{m-2}^{(t)} + \left(1 - \frac{\lambda}{2}\right)\mathbf{w}_{m-1}^{(t)} + \frac{\lambda}{2}hg(t) \tag{B.15}$$

que já sugerem qual é a adaptação a ser realizada. Os vetores $\mathbf{w}^{(t)}$ e $\mathbf{w}^{(t+1)}$ devem ter uma dimensão extra no índice m para acomodar $hg(t)$ e $hg(t+1)$, respectivamente. Como consequência, as matrizes A e B terão uma linha e uma coluna adicionais. As penúltimas linhas passam a ser definidas por:

$$a_{(m-1)k} = \begin{cases} 1 + \frac{\lambda}{2} & \text{se } m-1 = k \\ -\frac{\lambda}{2} & \text{se } |m-1-k| = 1 \\ 0 & \text{se } |m-1-k| > 1 \end{cases}$$

$$b_{(m-1)k} = \begin{cases} 1 - \frac{\lambda}{2} & \text{se } m-1 = k \\ \frac{\lambda}{2} & \text{se } |m-1-k| = 1 \\ 0 & \text{se } |m-1-k| > 1 \end{cases}$$

e a última linha de ambas as matrizes por:

$$A^{(m)} = B^{(m)} = \mathbf{e}^{(m)}$$

Portanto, o problema da equação do calor com critério de Dirichlet na fronteira inicial e critério de Neumann na fronteira final corresponde à igualdade $A\mathbf{w}^{(t+1)} = B\mathbf{w}^{(t)}$, onde a matriz A possui o formato

$$A = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\frac{\lambda}{2} & 1+\lambda & -\frac{\lambda}{2} & \cdots & 0 & 0 & 0 \\ 0 & -\frac{\lambda}{2} & 1+\lambda & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1+\lambda & -\frac{\lambda}{2} & 0 \\ 0 & 0 & 0 & \cdots & -\frac{\lambda}{2} & 1+\frac{\lambda}{2} & -\frac{\lambda}{2} \\ 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix}$$

e a matriz B possui o formato

$$B = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ \frac{\lambda}{2} & 1-\lambda & \frac{\lambda}{2} & \cdots & 0 & 0 & 0 \\ 0 & \frac{\lambda}{2} & 1-\lambda & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1-\lambda & \frac{\lambda}{2} & 0 \\ 0 & 0 & 0 & \cdots & \frac{\lambda}{2} & 1-\frac{\lambda}{2} & \frac{\lambda}{2} \\ 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix}$$

Os resultados serão consistentes se for possível garantir que um mesmo vetor $\mathbf{w}^{(t)}$ não-nulo está associado a um único vetor $\mathbf{w}^{(t+1)}$. Em outras palavras, se apenas o vetor zero está nos espaços nulos de A e B. Essa última afirmação equivale a dizer que as matrizes A e B possuem matriz inversa.

Uma condição suficiente para que exista a inversa de uma matriz é a *dominância diagonal estrita*, caracterizada pela desigualdade:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \tag{B.16}$$

Fora os casos explícitos da primeira e última linha em cada matriz, constata-se que a restrição imposta por (B.16) depende do valor de λ em todas as outras linhas.

A matriz A possui dominância diagonal estrita se $0 < \lambda < 2$. Já a matriz B impõe um valor máximo mais baixo para λ, que deve ficar no intervalo $0 < \lambda < \frac{1}{2}$. Por definição, $\lambda = \frac{k\alpha}{h^2}$, logo:

$$\left.\begin{array}{r} \lambda = \frac{k\alpha}{h^2} \\ 0 < \lambda < \frac{1}{2} \end{array}\right\} \implies 0 < k < \frac{h^2}{2\alpha} \tag{B.17}$$

A escolha de um valor para k dentro do intervalo (B.17) garante que tanto A quanto B possuem inversa. Logo, dado um vetor $\mathbf{w}^{(t)}$, é possível calcular o vetor no instante posterior $\mathbf{w}^{(t+1)}$ a partir da inversa de A:

$$\mathbf{w}^{(t+1)} = A^{-1}B\mathbf{w}^{(t)}$$

A matriz A^{-1} pode ser obtida por simples eliminação de *Gauss-Jordan*. Outro benefício da dominância diagonal estrita é a estabilidade numérica do processo de inversão, pois decorre desse fato que os autovalores de A possuem baixa variância.

Apêndice C

Códigos Fonte

Todos os códigos utilizados no trabalho estão disponíveis no repositório hospedado no *Github*:

https://github.com/lcnsqr/grnn
Implementação da GRNN para estimar a solução da equação do calor e do conjunto Mandelbrot. Há três implementações distintas: Sequencial em CPU, paralela em CPU (pthreads) e paralela em GPU. Também contém o código em Python para a rede neural do TensorFlow utilizada no comparativo.

Para compilar todos os comandos, executar `make all`. Para compilar os comandos individualmente:

- Gerador dos conjuntos para a equação do calor: `make geradorDifusao`
- Gerador dos conjuntos para o conjunto Mandelbrot: `make geradorDifusao`
- Estimador paralelizado em CPU com pthreads: `make grnn_pthreads`
- Estimador paralelizado em GPU: `make grnn_gpu`

Pode ser necessário alterar algumas definições no arquivo `Makefile` antes de compilar a versão GPU com Cuda, como a localização das bibliotecas Cuda (linha 37): `CUDA_PATH ?= "/usr/local/cuda"`.

C.1 Utilização

Sem nenhum argumento, os comandos `grnn_pthreads` e `grnn_gpu` apenas estimam o erro das estimativas para o conjunto de teste. Os conjuntos de treinamento e teste são gerados pelo comando `geradorDifusao` ou `geradorMandelbrot`. São criados os arquivos `train.bin` e `test.bin`, que serão utilizados pelos comandos `grnn_pthreads` e `grnn_gpu` para computar as estimativas.

C.1.1 Conjunto Mandelbrot

Estimativa de pertecimento de um ponto ao conjunto fractal Mandelbrot. São necessárias as bibliotecas de desenvolvimento *SDL2* para compilar esses comandos.

O comando `geradorMandelbrot` abre um tela com a representação gráfica do conjunto Mandelbrot. O parâmetro `-w 700` define a largura em `700` pixels e o parâmetro `-h 700` define a altura em `700` pixels. A quantidade de pontos no cnnjunto de treinamento corresponde à quantidade de pixels. A tecla `z` aproxima, a tecla `x` afasta e as teclas de direção transladam o semiplano. A tecla `s` armazena em `train.bin` o conjunto de treinamento (o semiplano exibido) e em `test.bin` o conjunto de teste (8192 pontos aleatórios dentro do semiplano exibido). Com a opção `-s` os conjuntos são gerados sem interatividade.

C.1.2 Equação do Calor

O comando `geradorDifusao` tem três opções: `-t`, `-e` e `-d`. A opção `-t TOTAL` define em `TOTAL` a quantidade de amostras de treinamento e a opção `-e TOTALTEST` define em `TOTALTEST` a quantidade de amostras de teste, onde `TOTAL` e `TOTALTEST` são números inteiros maiores que zero.

A opção `-d` define a quantidade de pontos igualmente espaçados no intervalo da solução, o que também determina o número de dimensões das variáveis independentes e dependentes da regressão.

C.1.3 Gerador das estimativas

Os comandos geradores das estimativas `grnn_pthreads` e `grnn_gpu` têm duas opções: `-o` e `-s`.

Um arquivo para armazenar as estimativas pode ser informado com a opção `-o result.bin` para que as estimativas sejam armazenadas no arquivo `result.bin`.

Um escalar para o parâmetro sigma da regressão pode ser informado com a opção `-s ESCALAR`, onde `ESCALAR` é um valor real positivo. O valor padrão é 1.

Bibliografia

[1] Specht, Donald F: *A general regression neural network*. IEEE transactions on neural networks, 2(6):568–576, 1991.

[2] Parzen, Emanuel: *On estimation of a probability density function and mode*. The annals of mathematical statistics, 33(3):1065–1076, 1962.

[3] Cacoullos, Theophilos: *Estimation of a multivariate density*. Annals of the Institute of Statistical Mathematics, 18(1):179–189, 1966.

[4] Sarle, Warren S: *Neural networks and statistical models*, 1994.

[5] Poggio, Tomaso e Federico Girosi: *A theory of networks for approximation and learning*. Relatório Técnico, Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1989.

[6] Narendra, Kumpati S e Kannan Parthasarathy: *Identification and control of dynamical systems using neural networks*. IEEE Transactions on neural networks, 1(1):4–27, 1990.

[7] Owens, John D, Mike Houston, David Luebke, Simon Green, John E Stone e James C Phillips: *Gpu computing*. Proceedings of the IEEE, 96(5):879–899, 2008.

[8] Ryoo, Shane, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk e Wen mei W Hwu: *Optimization principles and application performance evaluation of a multithreaded gpu using cuda*. Em *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, páginas 73–82. ACM, 2008.

[9] Harris, Mark: *How to optimize data transfers in cuda c/c++*. `http://web.archive.org/web/20180320132840/https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/`, 2012. Acessado: 2018-03-20.

[10] Harris, Mark: *How to overlap data transfers in cuda c/c++*. `http://web.archive.org/web/20180320134206/https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/`, 2012. Acessado: 2018-03-20.

[11] Harris, Mark: *How to access global memory efficiently in cuda c/c++ kernels*. http://web.archive.org/web/20180320134412/https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/, 2013. Acessado: 2018-03-20.

[12] Harris, Mark: *Using shared memory in cuda c/c++*. http://web.archive.org/web/20180320134614/https://devblogs.nvidia.com/using-shared-memory-cuda-cc/, 2013. Acessado: 2018-03-20.

[13] NVIDIA: *CUDA C PROGRAMMING GUIDE*. Relatório Técnico PG-02829-001, NVIDIA, August 2014. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, Design Guide v6.5, posted at 2014-08-01 13:24:15.

[14] Isaacson, Eugene e Herbert Bishop Keller: *Analysis of numerical methods*, capítulo 9, páginas 501–512. Dover Publications, 1994.